Computer Science Department

TECHNICAL REPORT

Detecting Nondeterminism in Shared Memory Parallel Programs

Anne Dinning

Technical Report 526

November 1990

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

Detecting Nondeterminism in Shared Memory Parallel Programs

Anne Dinning

Technical Report 526

November 1990

Detecting Nondeterminism in Shared Memory Parallel Programs

Anne Dinning

July 1990

A dissertation in the Department of Computer Science
Submitted to the Faculty of the Graduate School of Arts and Sciences
in partial fulfillment of the requirements for the degree of Doctor of Philosophy
at New York University

Approved: Bhubaneswar Mishra

Bhubaneswar Mishra
Research Advisor

Abstract

This thesis addresses the problem of detecting of a specific type of nondeterminism in shared memory parallel programs known as *access anomalies*. An access anomaly occurs when an update to a shared variable X is concurrent with either a read of X or another update of X.

The first part of the work considers dynamic detection of access anomalies. We introduce a new technique called *task recycling* that detects access anomalies "on the fly" by monitoring the program execution. This technique is designed with two goals in mind. The first goal is minimal monitoring overhead. Costs are incurred only at thread create, terminate, and coordinate operations and every time a monitored variable is accessed. Because variable accesses are generally the most frequent operation, the task recycling technique reduces the overhead per variable access to a small constant. The second goal is generality. The task recycling technique is applicable to a wide variety of parallel constructs and all common synchronous and asynchronous coordination primitives. Combined with a protocol for specifying ordering constraints, the method of representing concurrency relationships in task recycling can be extended to detect general race conditions in parallel programs.

The second part of the thesis involves static detection of several types of nondeterminism that makes dynamic anomaly detection inefficient. In particular, the notion of nondeterminism arising from critical section coordination is refined by distinguishing between three types of nondeterminism — *parallel*, *sequential*, and *reference* nondeterminism. The presence of these types of nondeterminism in a program impacts access anomaly detection in two significant ways: (i) how critical section coordination is modeled during anomaly detection, and (ii) the confidence level and complexity of guaranteeing that a program has no access anomalies. In particular, it is shown that access anomalies can be detected efficiently only if a program is parallel, sequential and reference deterministic. Heuristics are presented that make access anomaly detection tractable in the presence of other nondeterminism through a better classification and semantic understanding of a coordination protocol.

Acknowledgments

Although there are many people who helped me on the way to writing this thesis, I thank first and foremost my advisor Bud Mishra, and Edith Schonberg with whom many of the ideas in this thesis were developed. Working with them was both an enlightening and enjoyable experience. I am grateful to them for the generosity of their advice, patience and their friendship. I would like to express my appreciation to the other members of my committee: Allan Gottlieb, Dennis Shasha and Ed Schonberg. They have helped me not only at the end, but throughout my graduate school years. I would also like to thank the faculty in the Computer Science Department who gave me a foundation to work from and who were always available for helpful discussions, and the Ultracomputer research group for providing an environment for carrying out this research.

I am indebted to many other people. But in the effort to keep these acknowledgments short, I will thank jointly my fellow graduate students for their encouragement–especially Ben Bederson, Pasquale Caianiello, Naomi Silver and Ofer Zajicek–and my friends and family for their never-ending support.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

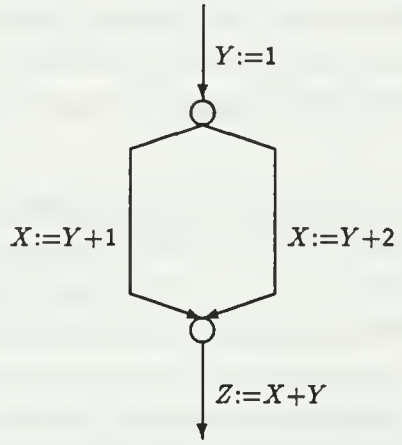
Introduction

The growth in popularity of multiprocessors in recent years is indicative of an increasing focus on achieving program speed-ups through parallelism. In order to decrease execution time, a program that executes on the multiprocessor must be a *parallel program*. A parallel program is a set of concurrently executing sequential processes that cooperate to solve a common problem. A particularly popular class of multiprocessors is general purpose multiple-instruction multiple data stream (MIMD) architectures with shared memory. Notable examples of this class of architecture include the Sequent, Alliant, Ultracomputer and Cedar multiprocessors [Seq89,PM86,KLC+86,Got88].

Shared memory is used in parallel programs to achieve coordination and synchronization. For example, writing to a shared memory location broadcasts information to other parallel threads, and signaling through a shared memory location synchronizes the execution of parallel threads. When shared memory is used for these purposes, concurrent access to the same memory location is expected. A completely different use of shared memory is for storing global shared data structures, such as large arrays in numeric code and tables and queues in systems software. The memory locations of these shared data objects are generally not intended to be accessed concurrently. For example, shared arrays in scientific parallel programs are often partitioned into sub-regions so that different threads can perform computations for each sub-region concurrently and asynchronously. However, when concurrent threads access the same element in the array and at least one access is a write operation, a coordination mechanism must be used to ensure correct access.

Erroneous behavior in shared memory parallel programs is often due to *access anomalies*. An access anomaly occurs when two concurrent execution threads access the same memory location in an "unsafe" manner: more specifically, when either two concurrent threads both write, or one reads and one writes a shared memory location without coordinating these accesses. This thesis examines the problem of access anomaly detection.

The program segment in Figure 1 illustrates the concept of an access anomaly. The doall construct creates two parallel threads that execute concurrently. Variable Y is




```
Y := 1
doall i := 1 to 2
    X := Y + i
endall
Z := X+ Y
```

Figure 1.1: Simple Program With an Access Anomaly

accessed by both concurrent threads. Because Y is only read concurrently, these accesses are safe. Similarly, the assignment to Y in the first statement does not cause an anomaly since this write is always performed before either read. However, because both concurrent iterates write the variable X, these accesses to X are anomalous. The final value assigned to X is either 2, 3 or 4 depending on the relative execution speed of the concurrent threads.

An access anomaly is a specific instance of a general type of behavior in parallel programs known as a *race condition*. A race condition exists when the execution of two operations that are not independent can occur in either order during the execution of a program. While certain race conditions do not affect the outcome of programs and are safe (e.g. the nondeterministic order in which a lock is granted), access anomalies often introduce nondeterminism that modifies execution results.

It is essential that access anomalies are detected and reported to the user or support environment. Perhaps the most important reason is that access anomalies are usually bugs. In fact, an Ada program that contains an access anomaly is defined to be erroneous. Although some programs are designed to contain this type of behavior, the nondeterminism stemming from access anomalies is usually unintentional. Access anomalies typically result from incorrect interthread coordination, as in missing lock operations, or program logic errors, as in incorrect array referencing.

Second, it is often useful to be able to repeat an execution of a parallel program for debugging purposes. It is not possible to reproduce a given execution instance E unless all race condition in E are detected and forced to occur in the same order in subsequent

re-executions. If access anomalies are not detected, trace-and-replay debuggers are of little use in debugging shared memory parallel programs.

Lastly, concurrent threads in a shared memory parallel program can communicate through either shared memory or coordination primitives. By detecting access anomalies, all interaction among concurrent threads is made explicit. Therefore, techniques that have been developed for analyzing and evaluating distributed memory programs can be applied to shared memory programs.

Unfortunately, traditional debugging techniques are of limited use in finding access anomalies, since such program behaviors are sensitive to execution timing. Several alternative detection methods have been proposed. One approach uses *static analysis* to isolate a set of potential access anomalies based on a static representation of the program. The primary benefit of static detection is that it provides general information about the program, rather than with respect to a given input vector. However, because static analysis is conservative, the number of potential anomalies reported may be very large. Inaccuracies stem from variable aliasing, pointers, and difficulties in determining if two portions of code can ever execute concurrently.

A second approach uses dynamic analysis to detect access anomalies in a single *execution instance* of a program. Dynamic analysis complements static analysis by locating anomalies precisely for that execution instance. In *trace-based* detection, all accesses to shared variables and all parallel operations are traced during program execution. This information is analyzed after the program completes. In *on-the-fly* detection of access anomalies, anomalies are detected as the program executes through the use of monitoring techniques. An access anomaly is discovered during program execution in much the same way as array subscript out-of-range exceptions are currently reported. As soon as an anomaly is detected, an exception is raised specifying the variable involved as well as the instruction counters. The primary advantages of on-the-fly detection over trace-based dynamic methods derive from data compression.

This thesis is organized as follows. Chapter 2 defines a *partial order execution graph* that represents the "happens-before" partial order relationship for an execution instance of a parallel program and describes how common thread creation, termination and coordination primitives are modeled in the partial order execution graph. Chapter 2 also gives a framework for determining when a single execution instance is sufficient for a dynamic anomaly detection algorithm to guarantee that a given program and input vector pair will never have an access anomaly.

Chapter 3 presents the *task recycling* technique, a new on-the-fly algorithm for detecting anomalies. Task recycling improves upon existing dynamic access anomaly detection techniques in three significant ways:

1. *Generality.* The task recycling technique is applicable to a wide variety of parallel constructs and all common synchronous and asynchronous coordination primitives.

2. *Performance.* Because variable accesses are generally the most frequent operation, the task recycling technique reduces the overhead per variable access to a small constant at the expense of a higher cost incurred at parallel operations.

3. *Storage.* The amount of information stored by the task recycling technique depends only on the maximum concurrency of the program, rather than the length of the program execution.

The approaches used in task recycling can be integrated into other parallel program debugging tools to improve their functionality.

Chapter 4 presents empirical measurements of on-the-fly detection as a general access anomaly detection technique and task recycling as a specific algorithm. Empirical data indicates that the task recycling algorithm performs better than existing on-the-fly techniques for a wide class of parallel programs. Moreover, the overhead incurred by task recycling is small enough to make it a viable tool for debugging parallel programs.

One of the primary drawbacks of dynamic detection is that it can find only those anomalies that occur in a single execution instance. In order to guarantee that there are no anomalies in a given program and input vector pair, one may have to examine many different execution instances. Chapter 5 presents a new representation for critical section coordination that decreases the number of execution instances that must be considered. Static analysis techniques are used to determine when this representation is guaranteed to accurately model the program behavior.

1.1 Existing Work

The notion of access anomalies was first defined in terms of *Read* and *Write* sets by Bernstein [Ber66]. Each sequence of instructions in a parallel program has a Read and a Write set associated with it: the Read set consists of those variables that are read in the instruction sequence and the Write set, those variables that are written. If two potentially concurrent instruction sequences S_i and S_j meet the following three conditions, all execution orderings of S_i and S_j are guaranteed to produce the same result (i.e. S_i and S_j do not introduce any external nondeterminism):

Bernstein's Conditions:

$$Read(S_i) \cap Write(S_j) = \emptyset$$
$$Write(S_i) \cap Read(S_j) = \emptyset$$
$$Write(S_i) \cap Write(S_j) = \emptyset$$

If these conditions are not met, concurrent execution of S_i and S_j may lead to access anomalies that alter the result of the execution.

In order to evaluate Bernstein's conditions, two types of information must be available:

1. Which instruction sequences potentially execute concurrently, and
2. Which variables are accessed by each instruction sequence.

One of the primary goals of an anomaly detection algorithm is to store and analyze this information as efficiently as possible.

1.1.1 Static Anomaly Detection

In static detection of access anomalies, a graph which represents the static structure of the program is analyzed and a set of potential access anomalies is reported. Traditional sequential control flow analysis techniques are used to determine the set of variables read and written in each node in the static program graph. Therefore, the primary issue in static detection of access anomalies is that of determining if two nodes can potentially execute concurrently.

One of the first static analysis algorithms for detecting access anomalies is due to Taylor [TO80,Tay83]. The primary emphasis in this work was the analysis of Ada programs; however, the techniques developed can be used to analyze any language with a rendezvous based coordination mechanism. An *annotated flow graph* is created for every Ada task. Because Taylor et al. are interested in intertask interaction, each node contains a summary of sequential operations performed by the code associated with that node. This technique of sequential abstraction is common to virtually all of the static representation techniques.

The analysis of annotated flow graphs is similar to symbolic execution. A *concurrency state graph* is generated by simulating all possible coordination sequences. Each node in the graph is a *concurrency state*, representing a set of tasks that can execute concurrently. Directed edges in the graph indicate which concurrency states can follow each other in a valid execution. Thus, the path from the root to a given state in the graph represents a valid *concurrency history* and describes one possible sequence of synchronization events which can occur. Access anomalies are then detected by analyzing the concurrency histories.

The strength of this approach is its accuracy. Instead of attempting to be conservative and summarizing information to worst case scenarios, all possible scenarios are considered and thus more exact answers are possible. This accuracy is essential for some issues in parallel program analysis (for example, the formal verification of Ada tasking programs [DKH88]). The primary weakness of this approach is that the number of possible concurrency states is exponential in the size of the program. In addition, it is very difficult

to model programs which dynamically create tasks, either with recursive procedures or pointers.

Appelbe and McDowell extend Taylor's work to parallel Fortran programs and use static analysis to generate input to a dynamic monitoring system [AM85,AM88,McD88]. Their work in static analysis of parallel programs is just one component of a system for generating multitasking application programs. Static analysis is performed by creating a standard control flow graph and then compressing it into a *synchronization* graph which contains only relevant synchronization operations. This graph is then used to create a *concurrency history graph*. They use the concurrency history graph to detect "synchronization anomalies" (states that lead to program deadlock) as well as access anomalies.

The third major body of work in this area is by Emrath and Padua. In [EP88], they present some preliminary ideas on the detection of nondeterminism in parallel programs. They define a *ordering graph* whose nodes represent statements, and whose arcs represent execution order between statement instances. Although they do not describe how ordering graphs are created or analyzed, they present a clean mechanism for representing coordination among parallel loop iterates.

Each coordination edge has an associated *distance vector* which is annotated with the distance between the two coordinating threads. This allows for a compact representation of the coordination among parallel loop iterates. Distance vectors are similar to the direction vectors which are used to represent data dependences in analysis of sequential programs. However, rather than simply containing $<, >$ or $=$ information, an entry in a distance vector contains the distance–measured in number of loop instances–of the two coordinating parallel loop instances.

PTOOL is an environment designed at Rice for developing and debugging both automatically parallelized and explicitly parallel Fortran programs [BKK+88,BKK+89]. One aspect of the system is a tool for detecting and inspecting potential access anomalies. Callahan and Subhlok [CS88,CKS90] propose a *synchronized flow graph*, based in part on the ideas proposed by Emrath and Padua, to detect potential anomalies. They focus on event synchronization and use instance and distance vectors analyzing coordination among loop iterates. In contrast to the preceding systems, a more general data flow approach is used which results in less exact, but computationally tractable, solutions.

A second effort in conjunction with the PTOOL environment is by Balasundaram and Kennedy [BK89]. They use a *co-graph* similar to the ordered graph of Padua and Emrath. Their primary emphasis is on detection of various types of deadlock conditions, although their analysis techniques can be used to detect access anomalies.

1.1.2 Trace-Based Anomaly Detection

Trace-based anomaly detection is a dynamic technique that finds the anomalies that occur in a given execution instance. In trace-based anomaly detection, all accesses to shared variables and all parallel operations are traced during program execution. After the program completes, a graph that represents the concurrency structure of the program execution is built, the concurrency relationship is computed, and a set of access anomalies is reported.

Choi, Netzer and Miller proposed a trace-based technique for dynamic access anomaly detection [MC88,CMN88,Cho89] as part of a large system for debugging parallel programs. A *before* and an *after* vector is associated with each instruction sequence S which contains information about which instruction sequences respectively p- -eded S and followed S during the execution of the program. Access anomalies are c -ected by comparing the accesses made by each instruction sequence S with the accesses by all instruction sequences that are concurrent with S. Netzer and Miller [NM89] developed methods for determining whether or not an access anomaly is a side-effect of anomalies that preceded it in the execution instance. When this is the case, additional ordering is added to the graph to eliminate this "false" anomaly.

Emrath, Ghosh and Padua [EGP89,EP88] have also developed a trace-based technique for detecting access anomalies. In contrast with the system of Choi and Miller, they attempt to model the ordering that is guaranteed to occur in similar execution instances, rather than what actually appeared in the given execution instance. The concurrency relationship is computed iteratively by finding the closest common ancestor of all candidate coordination partners. This relationship may be incorrect in that two instruction sequences that can never execute concurrently may be misidentified as being concurrent.

1.1.3 On-The-Fly Anomaly Detection

Similar to trace-based detection, *on-the-fly* anomaly detection finds the anomalies that occur within a given execution instance. However, in on-the-fly detection the program execution is monitored, and anomalies are detected as they occur. The primary advantages of on-the-fly detection over trace-based dynamic methods derive from data compression. Variable access and parallel operation information that is no longer needed is discarded as the program executes. Moreover, because anomalies are detected during execution, breakpoints or exception handling routines can be invoked when an anomaly is detected. However, fewer anomalies are reported here than in trace-based analyses, and sophisticated concurrency relationships among executing threads cannot be modeled.

The Merge algorithm [Sch89,Sni88] is derived from the following observation: if more than one concurrent instruction sequence reads the same variable, these read events may

be collapsed into a single event after all concurrent instruction sequences complete. To implement the Merge algorithm a *concurrency list* is associated with one or more instruction sequence, and Read and Write sets are associated with concurrency lists. The concurrency list associated with an instruction sequence S is the set of tasks T such that the instruction sequence currently executing in T is concurrent with S. When an instruction sequence in task T terminates, its Read and Write set are compared with all Read and Write sets associated with concurrency lists L_i that contain T, and T is deleted from each L_i. Read and Write sets are merged whenever their associated concurrency lists are equal. In the worst case, the number of concurrency lists–and therefore the number of Read and Write sets–is quadratic in the parallelism of the program.

Nudler and Rudolph [NR88a,NR88b] proposed a scheme known as *English-Hebrew labeling.* Read and write set information is stored in an inverse format by associating an access information with every shared variable X that is checked whenever X is read or written. Concurrency information is maintained by associating a tag with each instruction sequence that consists of a pair of labels: an *English label* E and a *Hebrew label* H. Two instruction sequences S_i and S_j with tags t_i and t_j are potentially concurrent if and only if the following condition is met:

$$(E(t_i) < E(t_j) \text{ and } H(t_i) > H(t_j)) \text{ or } (E(t_i) > E(t_j) \text{ and } H(t_i) < H(t_j))$$

English-Hebrew tags only partially encode the concurrency relationship. Therefore, each executing instruction sequence S has a *coordination list* that contains the tags of instruction sequences that occurs before S but fail the above test. The complete test for concurrency requires determining if the tag of S_i or any of the tags in the coordination list of S_i are ordered with the tag of S_j.

Hood, Kennedy and Mellor-Crummey are developing an on-the-fly detection algorithm for a subset of the Parallel Fortran language supported by PTOOL [HKMC89]. In particular, they support nested parallel constructs in which the only valid form of coordination is doacross coordination of distance 1. The goal of their technique is to maintain concurrency information efficiently by taking advantage of the regular concurrency relationship of programs in this class.

Concurrency information is maintained by associating a tag with each instruction sequence. The tag of an instruction sequence in iterate i of an unnested normalized doall loop is computed as follows:

$$tag = B + i + k$$

where k is the number of doacross operations that have been performed, N is the number of iterates in the doall, and B is the tag of the block that performed to doall operation. An instruction sequence in iterate i is a descendant of a block with tag tag if and only if

the following condition holds:

$$((tag - B - 1) \text{ div } N < (1 + k)) \text{ and } ((tag - B - 1) \text{ mod } N) < (1 + i)$$

For nested parallel constructs, a stack of tags are maintained and checked at every access. The maximum number of instruction sequences in an iterate is assumed to be known a priori or conservatively estimated using static analysis.

Steele [Ste90] proposes a technique for detecting anomalies in programs that do not coordinate. The structure of the concurrency relationship at any point in time is a tree; this regularity is used to determine whether or not two instruction sequences are concurrent. When an instruction sequence S accesses a variable V, it checks that all instruction sequences at *lower* levels in the concurrency tree T that also accessed V are on the path from S to the root of T. When an instruction sequence terminates, it checks all instruction sequences of higher nesting levels in T.

Chapter 2

Parallel Program Representation

A parallel program consists of a set of parallel threads that cooperate to solve a problem. During any given execution E of a parallel program, certain instruction sequences are constrained to execute consecutively due to sequential control flow constraints and the semantics of the thread creation, termination and coordination operations performed. Other instruction sequences, however, can execute at the same time. If one instruction sequence S_i is guaranteed to have executed before another instruction sequence S_j because of the parallel operations performed in an execution instance E, then S_i and S_j are ordered by the "happens-before" partial order relation as defined by Lamport [Lam78,Pra86].

The "happens-before" relationship of an execution instance E is used to detect access anomalies in E. Specifically, two accesses to the same shared variable are anomalous only if they are performed by two instruction sequences that are not ordered by this relation. Section 2.1 defines a *partial order execution graph (POEG)* that represents the "happens-before" relationship for an execution instance of a parallel program. Section 2.2 describes how the POEG is used to determine concurrency relationships among the instruction sequences.

The partial ordered relationship represented by a POEG is based on the semantics of the parallel operations in the program; it does not reflect sequential execution that occurs as a side effect of scheduling decisions or due to the parallelism of the underlying machine. However, it is crucial that all of the instruction sequence interaction in a given execution instance is correctly modeled in the associated POEG. If the relationship modeled by the POEG is incorrect, the accuracy of anomaly detection is affected: false anomalies can be reported and/or access anomalies can be hidden in many or all execution instances. Section 2.3 discusses how common coordination primitives are accurately represented in a POEG.

In general, many different execution instances of the same program and input vector pair are represented by the same POEG. However, two execution instances of a given

program and input vector pair cannot always be mapped to the same POEG. When this is the case, anomalies may go undetected in many execution instances. This makes it difficult to say that a given program and input vector pair does not have any access anomalies. Section 2.4 discusses the notion of *reliability* in dynamic anomaly detection and presents results on the complexity of guaranteeing that a program will never have an access anomaly when executed on a given input vector.

2.1 Structure of the Partial Order Execution Graph

A partial order execution graph represents all interaction among instruction sequences through parallel operations. Vertices in a POEG are associated with parallel operations: namely, *fork, join* and *coordination vertices.* Two types of directed edges represent the sequential and parallel control flow constraints. A *block edge* of a POEG represents an instruction sequence and is delimited by parallel operations; a *coordination edge* (denoted by a dashed line) connects two coordination vertices. Thus, a partial order execution graph is represented by a triple: vertices, block edges and coordination edges.

In parallel programs, parallelism is achieved through a thread creation or *fork* operation that creates several threads that execute concurrently. Similarly, a thread termination or *join* operation indicates a decrease in parallelism. Fork and join operations are represented in the POEG by *fork* and *join* vertices:

- A *fork vertex* has one in-edge (the block that performed the fork operation) and f out-edges (the f blocks created by the fork operation).
- A *join vertex* has one out-edge (the block that follows the join operation) and j in-edges (the j blocks terminated by the join operation).

The basic control flow for a parallel program is defined by the fork and join operations as well as the sequential control flow within a given block edge. The execution of a block that performs a fork operation must precede the execution of all of the threads created in the operation. The block that executes after a join operation can begin only after all blocks terminating in the join operation are complete.

A *parallel construct* is a closed, nestable mechanism for creating parallel threads and consists of a fork and a join operation. The semantics of a parallel construct requires that exactly those threads created by the fork operation terminate at the associated join operation. Examples of parallel constructs include doall and parallel case constructs. (A doall construct creates a set of n homogeneous thread iterates, a parallel case construct creates heterogeneous threads.) The POEG for a program that only contains parallel constructs is a series-parallel graph. A series-parallel POEG allows for certain optimization in representing and analyzing the graph during program execution. Unless otherwise stated, we assume that threads are always created using parallel constructs.

Figure 2.1 illustrates how a simple parallel program is represented by a POEG. Every

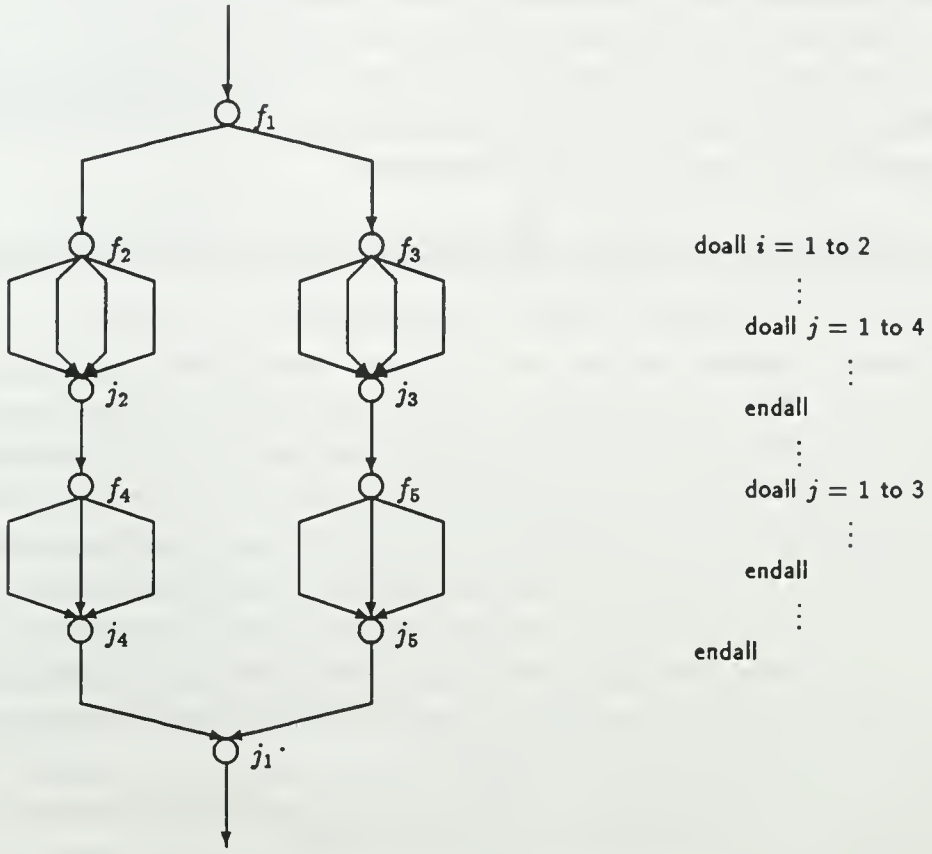


Figure 2.1: POEG for Nested Doall Program

doall operation has corresponding fork and join vertices. Each increase in parallelism is represented by a fork vertex–f_1, f_2, f_3, f_4, and f_5–and each decrease in parallelism is represented by a join vertex–j_1, j_2, j_3, j_4, and j_5. Because a doall is a parallel construct, all threads created by a fork vertex terminate at the associated join vertex (for example, f_1 and j_1).

Programmers use synchronization and coordination to control access to shared data or pass status information. Therefore, a POEG must also reflect the implicit and explicit orderings that results from this interaction. As a practical matter, it is not always possible to recognize when shared memory is used for coordination. For example, a thread can use a shared variable to broadcast information to other threads. If this is not identified as a coordination operation, the concurrent accesses to shared memory will be improperly identified as access anomalies. More importantly, the resultant ordering will not be added to the POEG. Only coordination that is explicitly specified by the programmer through

the use of identifiable routines is represented in a POEG; the complexity of correctly detecting implicit coordination as well as deducing the intended ordering is beyond the scope of this thesis.

One type of ordering that is represented in the POEG is the *explicit* control flow ordering that results from coordination. For example, the semantics of the barrier coordination primitive require that a block that executes after a barrier operation in one thread must follow the execution of all blocks that coordinate in the barrier.

The nondeterministic execution order of certain coordination operations can result in *implicit* ordering among blocks. For instance, the semantics of critical section coordination does not specify an explicit ordering among threads that execute the critical sections. However, the execution order of the critical sections can affect subsequent execution. In addition, certain concurrent accesses to shared variables may be intentional; these accesses are treated as a special type of *shared memory coordination.* The implicit order that stems from coordination must be represented in the POEG whenever the result of program execution depends on the order in which the blocks coordinate.

One could imagine coordination primitives that enforce arbitrary ordering constraints among mutually concurrent blocks. In general, however, the goal of coordination is to synchronize the execution of several blocks at some coordination point. Therefore, we restrict the notion of coordination within an execution instance as follows:

> **Coordination Constraint:** A coordination point is a stage P during the execution of a program such that one set of blocks R can continue execution only if a second set of blocks S has reached P.

Thus, if a block is waiting for a set of blocks S to reach some point in their execution, any other block that is waiting for any block in S must wait for all blocks in S. This coordination constraint bounds the complexity of algorithms presented in Chapter 3 while remaining general enough to represent all common forms of coordination. (The representations presented in Section 2.3 meet this constraint.) For the remainder of this thesis, all coordination is assumed to meet this constraint.

The implicit and explicit ordering that arises from a coordination point is represented in the POEG by adding coordination edges that connect coordination vertices, thereby ordering the subsequent execution. A coordination vertex can be one of two types:

- A *sender vertex* has n coordination out-edges to associated receiver vertices.
- A *receiver vertex* has n coordination in-edges from associated sender vertices.

In addition, every coordination vertex has an in-edge which is the block preceding the coordination operation and an out-edge which is the block following the coordination operation. Synchronous coordination is represented by a set of sender and receiver vertices

in each coordinating thread (for an example see Section 2.3.1) and is generally shown in POEG diagrams as a bidirectional coordination edge.

To illustrate the use of coordination edges, consider the program fragment in Figure

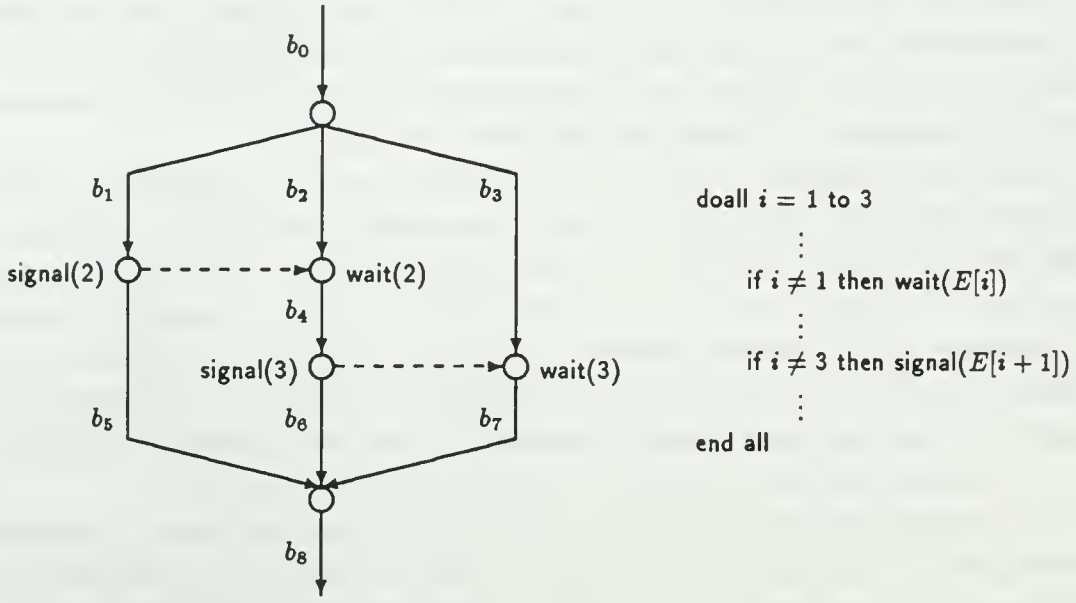


Figure 2.2: POEG with Signal-Wait Coordination

2.2. The semantics of signal-wait coordination requires that the execution of a signal operation precede the execution of the associated wait operation. This ordering is modeled in the POEG by a coordination edge from a sender vertex that represents the signal operation, to a receiver vertex that represents the associated wait operation. The coordination edge denotes that all of the execution that happened before the signal operation is guaranteed to have completed before all of the computation that followed the wait operation. Thus, the execution of block b_1 must precede that of block b_4 and the execution of block b_4 must precede that of block b_7. Because the ordering relationship is transitive, the execution of block b_1 must also precede block b_7.

2.2 Detecting Concurrency Using the POEG

The "happens-before" partial order relation is globally captured by the interblock relationships defined below:

Definitions:

A block a is an *ancestor* of a block b (and block b is a *descendant* of block a) if and only if there is a path from a to b in the POEG.

A block a is a *direct ancestor* of a block b if and only if there is a path from a to b in the POEG that does not include a coordination edge.

A block a is an *indirect ancestor* of a block b if and only if all paths from a to b in the POEG include a coordination edge.

The *direct parents* (resp. *indirect parents*) of a block b is the set of closest direct ancestors (resp. indirect ancestors) of b.

Two blocks are *concurrent* if and only if neither is an ancestor of the other.

The *maximum concurrency* of a POEG is the maximum number of mutually concurrent blocks.

A block a can be an ancestor of a block b either from sequential control flow constraints (e.g. because of block edges) or due to the ordering imposed by coordination edges.

To illustrate these definitions, consider the POEG in Figure 2.3. This POEG represents a program segment containing two nested parallel constructs and an asynchronous coordination operation between blocks b_2 and b_6. Block b_2 is the direct parent of block b_8;

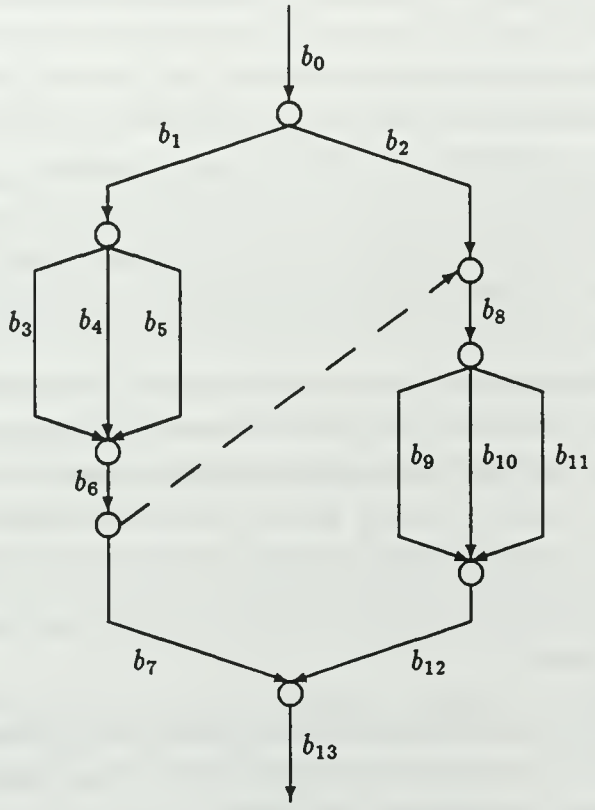


Figure 2.3: POEG with Coordination

block b_6 is the indirect parent of b_8 because of the coordination edge between blocks b_2 and

b_6. Block b_3 is concurrent with blocks b_4, b_5, and b_2; it is not concurrent with block b_0 or b_1 (which are ancestors), with blocks b_6, b_7, or b_{13} (which are direct descendants), or with blocks b_8, b_9, b_{10}, b_{11} or b_{12} (which are indirect descendants). The maximum concurrency of the POEG in Figure 2.3 is four, since blocks b_3, b_4, b_5 and b_2 are mutually concurrent (as are blocks b_7, b_9, b_{10} and b_{11}).

Theorem 1 proves that this definition of concurrent execution agrees with the common concept of concurrency:

> **Theorem 1:** Two blocks b_i and b_j in a POEG can execute at the same time if and only if b_i is neither an ancestor nor a descendant of b_j.

The advantage of a POEG representation is that concurrency determination is independent of the number and relative speed of processors executing the program. This is because the partial order relationship expressed by the POEG is based on the semantics of a program rather than the actual execution time of instruction sequences. By more accurately reflecting the causal ordering of the program, the significance of anomaly detection results is increased, since these results are independent of the underlying architecture. Moreover, this allows access anomalies in parallel programs to be detected on a uniprocessor.

Because coordination introduces additional ordering, the maximum concurrency of a POEG with coordination edges may be less than the maximum concurrency of the same graph with coordination edges removed. For instance, the maximum concurrency of the POEG in Figure 2.3 is increased from four to six if the coordination edge between b_2 and b_6 is removed. Intuitively, since adding coordination edges reduces concurrency, one expects anomaly detection to be more efficient for programs that coordinate. However, the partial order relationships in POEG with coordination edges are more difficult to encode efficiently. This added complexity is reflected in increased computational and space requirements for managing concurrency information in anomaly detection algorithms.

Asynchronous coordination introduces two additional problems. First, a POEG with coordination could have a cycle of arbitrary length. However, Theorem 2 shows that anomaly detection in cyclic graphs is hidden by the larger problem of deadlock.

> **Theorem 2:** A set of edges C form a cycle in a POEG if and only if the blocks in C are deadlocked.

Second, anomaly detection methods require that the concurrency information of all coordinating blocks be available when the coordination occurs. Thus, whenever a sender block reaches an asynchronous coordination point before the associated receiver block, information about the sender must be saved until the receiver reaches the coordination point. This leads to additional cost for buffering concurrency information for asynchronous coordination.

Theorem 1 *Two blocks b_i and b_j in a POEG can execute at the same time if and only if b_i is neither an ancestor nor a descendant of b_j.*

Proof. Let $create_b$ and $term_b$ denote the creation and termination times respectively of block b in a POEG and the *relatives* of b denote all of the ancestors of b and itself.
$\Longrightarrow$ Suppose, for the sake of contradiction, that there is a block b_i that can execute at the same time as another block b_j, and b_i is an ancestor of b_j. Because block b_j is a descendant of b_i, b_j can only be created after b_i has terminated. Therefore, $term_{b_i} \leq create_{b_j}$. However, for two blocks to be able to execute at the same time, it must be possible for both of their creation times to precede either termination times. Therefore, b_i cannot execute at the same time as b_j.
$\Longleftarrow$ Suppose, for the sake of contradiction, that there are two blocks b_i and b_j such that b_i is neither an ancestor nor descendant of b_j, and b_i cannot execute at the same time as b_j. Let a_i and a_j be the closest non-common relatives of b_i and b_j such that a_i can execute at the same time as a_j. There must be such a pair of blocks: they are two children of the closest common ancestor of b_i and b_j. Consider three cases for the relationships of b_i, a_i, b_j and a_j:

1. $b_i = a_i$ and $b_j = a_j$: It follows directly that b_i can execute at the same time as b_j.

2. $b_i \neq a_i$ and $b_j \neq a_j$: Block a_i cannot execute at the same time as any children of a_j (because they are all closer relatives of b_j than a_j). Therefore $term_{a_i} \leq term_{a_j}$. Likewise, $term_{a_j} \leq term_{a_i}$ and hence $term_{a_i} = term_{a_j}$. The only way to enforce this equality is if a_i and a_j synchronize, which makes a_i and a_j relatives of both b_i and b_j.

3. $b_i = a_i$ and $b_j \neq a_j$ (or vise versa): We know $term_{b_i} \leq term_{a_j}$. To enforce this, b_i (or one of its descendants) must be a parent of a_j (or one of its ancestors) and hence b_i is an ancestor of b_j.

In all cases we get a contradiction and therefore b_i must be able to execute at the same time as b_j. □

Theorem 2 *A set of edges C form a cycle in a POEG if and only if the blocks in C are deadlocked.*

Proof.
$\Longrightarrow$ Suppose, for the sake of contradiction, that there is a POEG with edges from b_{i+1} to b_i for $1 \leq i \leq n$, an edge from b_1 to b_n and there exists some order of execution in which blocks $b_1 \ldots b_n$ all terminate. For $1 \leq i \leq n$, b_i cannot begin until b_{i+1} terminates

and therefore must terminate strictly after b_{i+1} terminates; thus, $term_{b_1} > \ldots > term_{b_n}$. However, the edge from b_1 to b_n requires $term_{b_n} > term_{b_1}$ which is a contradiction.
$\Leftarrow$ Suppose, for the sake of contradiction, that some blocks are deadlocked and there is no cycle in the POEG. There must exist at least one block b_i in the deadlock whose ancestors $b_{i_1} \ldots b_{i_j}$ are not in the deadlock (the root block can always execute). At some point blocks $b_{i_1} \ldots b_{i_j}$ will terminate (since they are not deadlocked) and b_i can terminate. Hence b_i could not have been deadlocked. □

2.3 Representing Coordination Operations

It is essential that the ordering that results from the execution of a parallel operation is correctly represented in a POEG. If the ordering represented by the POEG is incorrect, the accuracy of anomaly detection can be affected in two ways. First, if the order described by a POEG is a subset of the partial order relationships intended by the programmer, two accesses that cannot execute concurrently may be incorrectly identified as an access anomaly. Second, if the ordering among instruction sequences described by a POEG is a superset of the intended partial order relationship, access anomalies may go undetected.

There are many different coordination primitives, and a POEG representation must be individually designed for each primitive [Din89], [AS83]. The primary difficulty in representing a coordination primitive lies in correctly capturing its effect on program execution. Some primitives have complex ordering semantics. Moreover, all of the relevant information may not be available at the time that certain blocks reach a coordination point. This section describes how several common coordination primitives–namely, barrier coordination, critical sections, message passing, doacross coordination, Ada rendezvous and events–are represented in the POEG. The techniques used here can be applied to other coordination primitives.

2.3.1 Barrier Coordination

An n-way barrier coordination synchronizes the execution of n threads. Semantics of a barrier are that each thread executing a barrier must wait until all threads have performed the barrier, at which point all may continue their execution [Axe86]. It is incorrect for either $n-1$ or $n+1$ concurrent threads to execute the same n-way barrier. Barriers are commonly used in parallel programs that execute in phases. All threads execute a barrier at the end of phase i, thereby guaranteeing that all processing of phase i is complete before any processing of phase $i+1$ is begun.

Barrier synchronization among threads $t_1 \ldots t_b$ is represented in the POEG by $2\times(b-1)$ coordination edges. Each thread t_i is modeled as performing a sequence of four steps:

1. Asynchronously coordinating with thread t_{i-1} as a receiver

2. Asynchronously coordinating with thread t_{i+1} as a sender

3. Asynchronously coordinating with thread t_{i+1} as a receiver

4. Asynchronously coordinating with thread t_{i-1} as a sender

Thread t_1 does not perform steps 1 and 4, and thread t_b does not perform steps 2 and 3. Similar techniques are used to represent any form of synchronous coordination.

To illustrate this representation, Figure 2.4 shows the POEG that results when a 4-

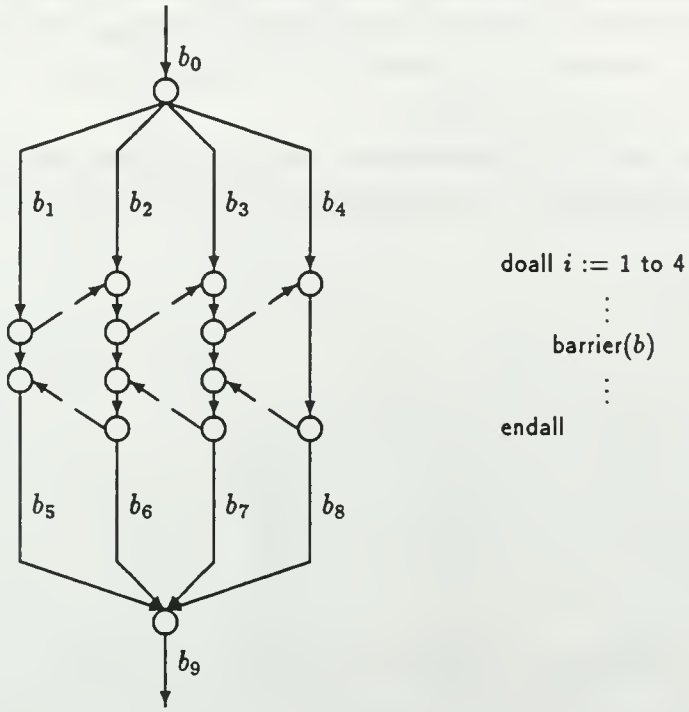


Figure 2.4: POEG with Barrier Coordination

way barrier is executed by blocks b_1, b_2, b_3 and b_4. As a result of the barrier operation, all blocks in phase 1–b_1, b_2, b_3 and b_4–are ancestors of all blocks in phase 2–b_5, b_6, b_7 and b_8.

2.3.2 Message Passing Coordination

In message passing coordination, one block sends a message that another block receives. Message passing is most often used in distributed memory programs. However, its semantics is general enough so that message passing can be used to model many other coordination primitives. For instance, signal-wait coordination corresponds to asynchronous message passing in which all messages are null.

The coordination that results from the exchange of a message between a sender thread t_S and a receiver thread t_R depends on the type of message passing. In *asynchronous* message passing, the sending block does not wait for the receiving thread to actually receive the message. The sender may proceed as soon as it buffers the message for the receiver. However, the receiver must wait for the message to arrive. Thus, asynchronous message passing is represented by a sender vertex in t_S connected to a receiver vertex in t_R.

In *synchronous* message passing, the sender and receiver threads must wait for each other before a message can be passed between them. Therefore, both threads must be at the coordination point at the same time. This is represented in the POEG as a sender vertex in t_S connected to a receiver vertex in t_R, followed by a sender vertex in t_R connected to a receiver vertex in t_S.

Figure 2.5 shows the POEG representation for two programs with asynchronous and synchronous message passing. In both programs block b_1 sends a message to block b_2. In

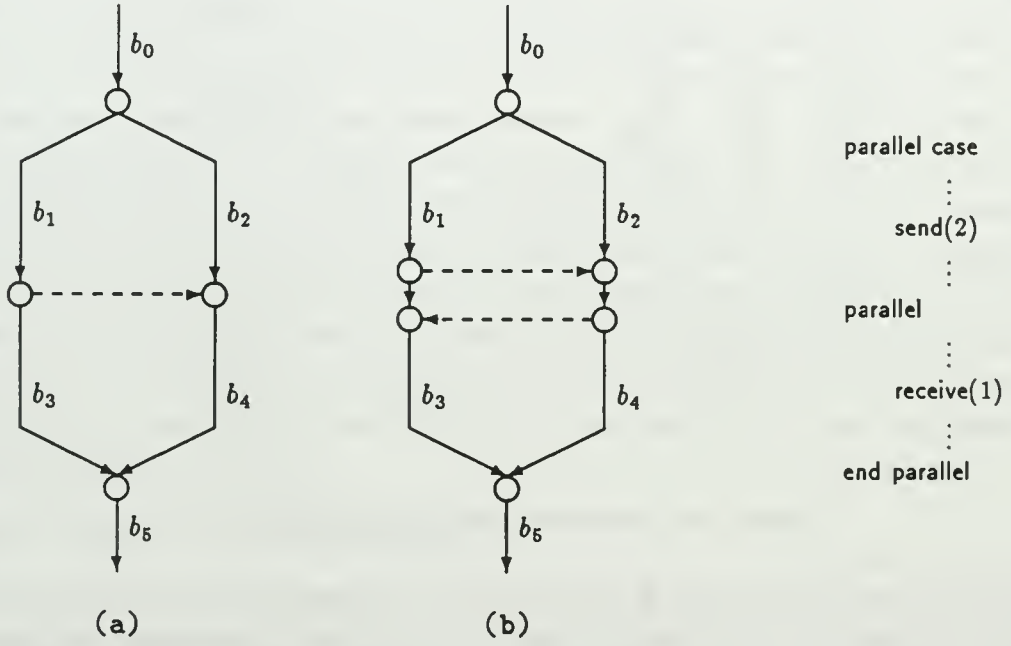


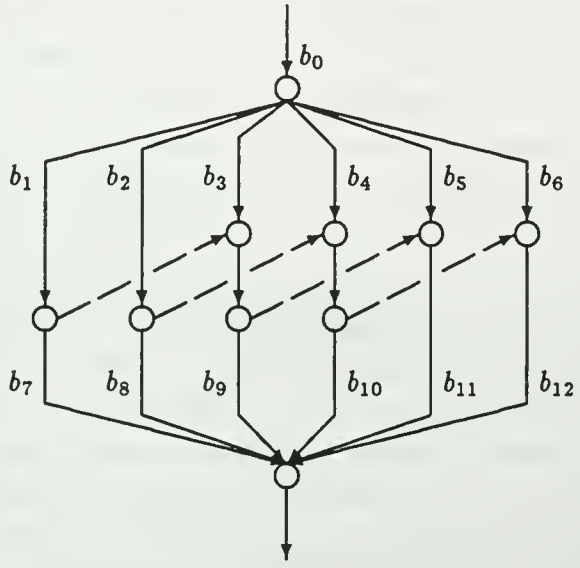
Figure 2.5: POEG with (a) Asynchronous and (b) Synchronous Message Passing

the case of asynchronous message passing, block b_3 can proceed before block b_2 reaches the coordination point. Therefore, block b_3 is concurrent with both blocks b_2 and b_4. Block b_4 must wait for block b_1 to send the message, and hence it is a descendent of b_1. For synchronous message passing, both blocks b_3 and b_4 can proceed only after both b_1 and b_2 have reached the coordination point. Therefore, b_1 and b_2 are ancestors of both b_3 and b_4.

2.3.3 Doacross Coordination

Doacross coordination is a structured form of asynchronous message passing used in doall constructs [Cyt84]. In doacross coordination, a thread t_i must wait at some point w in its execution until another thread t_{i-d} reaches a point s in its execution, where d is a fixed iterate distance. This is modeled in a POEG as thread t_{i-d} asynchronously sending a message to thread t_i.

The use of doacross coordination leads to a pipelined style of execution. Figure 2.6 shows a program with a doacross with distance 2. Each iterate t_i receives a message from




```
doall i := 1 to 6
        ⋮
    receive(i)
    A[i] := f(A[i − 2])
    send(i + 2)
        ⋮
endall
```

Figure 2.6: POEG with Doacross Coordination

iterate t_{i-2} and then uses the entry of A that was computed by that iterate. As a result of the doacross coordination, blocks b_2 and b_4 are indirect ancestors of block b_{12} (blocks b_0 and b_6 are direct ancestors of block b_{12}). Block b_{12} is not ordered with any block in iterates 1, 3 and 5.

2.3.4 Ada Rendezvous

The Ada rendezvous is a special type of asynchronous message passing known as a *remote procedure call* [Sta83]. In an Ada rendezvous, a *calling* block requests a *called* block to perform some remote action. The calling block immediately waits for the called block to send the result of the action. After the called block completes the remote action, it returns the result to the calling block or indicates that the request has been completed (if the procedure does not have any results).

An Ada rendezvous is composed of a pair of message exchanges. The first message is sent from the calling thread to the called thread, requesting it to execute a remote procedure. The second message is sent from the called thread to the calling thread after the execution of the remote procedure and contains the result of the execution. These messages are represented by two asynchronous coordination edges. Because the calling thread begins waiting for the result immediately after it makes the request, there is conceptually a single coordination vertex in the calling block.

Figure 2.7 shows a rendezvous between two parallel threads. Calling block b_1 in task

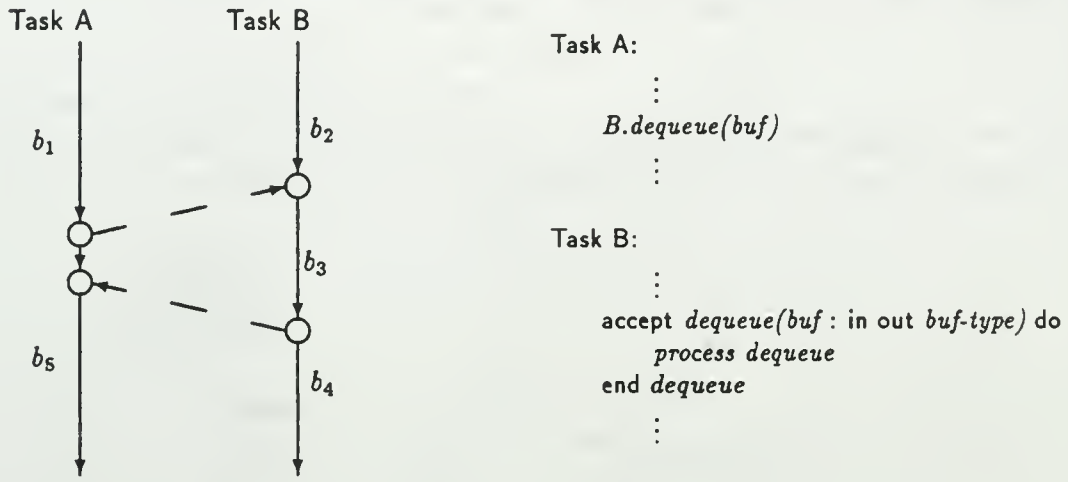


Figure 2.7: POEG with Ada Rendezvous

A requests the called block b_2 in task *B* to perform a *dequeue* operation. This results in the first coordination edge in the POEG from block b_1 to b_2. Block b_3 performs the *dequeue* operation and sends the result to block b_1 (via the in out parameter *buf*). This creates the second coordination edge from block b_3 to b_1.

2.3.5 Critical Section Coordination

A *critical sections* is an instruction sequence that protects access to shared variables by guaranteeing that all critical sections are executed sequentially [Dij65,Dij68,Dij71]. Lock and unlock operations delimit critical sections, as illustrated by the program fragment in Figure 2.8.

Two properties must be captured by the POEG representation of critical section coordination:

1. Accesses made inside critical sections never conflict.

2. An access made inside a critical section and an access made outside a critical section can conflict.

In addition, the unpredictable order in which parallel threads execute critical sections can affect the result of execution. For example, in the program fragment shown in Figure 2.8, the lock operations ensure that no two iterates update X at the same time; however, the order in which updates occur is arbitrary and determines the final value of X.

Critical section operations are modeled in the POEG by treating the unlock operation as a send operation and the lock operation as a receive operation. This results in a coordination edge from the block that performed the i^{th} unlock operation to the block that performs the $i+1^{st}$ lock operation; this edge orders all accesses made in the i^{th} and $i+1^{st}$ critical section executions. Because all critical sections are ordered in the POEG, no anomalies will be incorrectly reported among accesses performed within critical sections. These edges reflect all interaction–and therefore all implicit ordering–among parallel threads.

To illustrate the representation of critical section coordination, consider the program fragment and POEG in Figure 2.8. The execution instance illustrated by the POEG in Figure 2.8 is one in which iterates i_1, i_3 and i_2 execute the critical section, in that order. The coordination edge between i_1 and i_3 prevents the write accesses to X in the critical section by iterates i_1 and i_3 from being incorrectly identified as access anomalies. Since all implicit ordering is represented, the accesses to B are not falsely reported as access anomalies. (Control to the entries in array B is passed through the critical section.)

Anomalies can be masked when the critical sections execute in a given order. Thus, multiple execution instances must be analyzed to find all access anomalies. In particular, $N!$ POEGs are required to model the different execution orders of N concurrent critical sections. For example, there are six possible execution orders of the critical section in the program in Figure 2.8. In the execution instance shown, the read and write events to $A[1]$ are also ordered by the coordination edges added to the POEG and hence no write-read access anomaly will be detected in this execution instance. However, these accesses to A will be correctly identified as an access anomaly in execution instances in which i_3 enters the critical section before i_1. Chapter 5 addresses some of the weaknesses of this representation.

2.3.6 Event Coordination

Events are a type of coordination used in parallel Fortran programs for interthread signaling. An event e has two possible states, *posted* and *cleared*, and three operations can be performed on an event e:

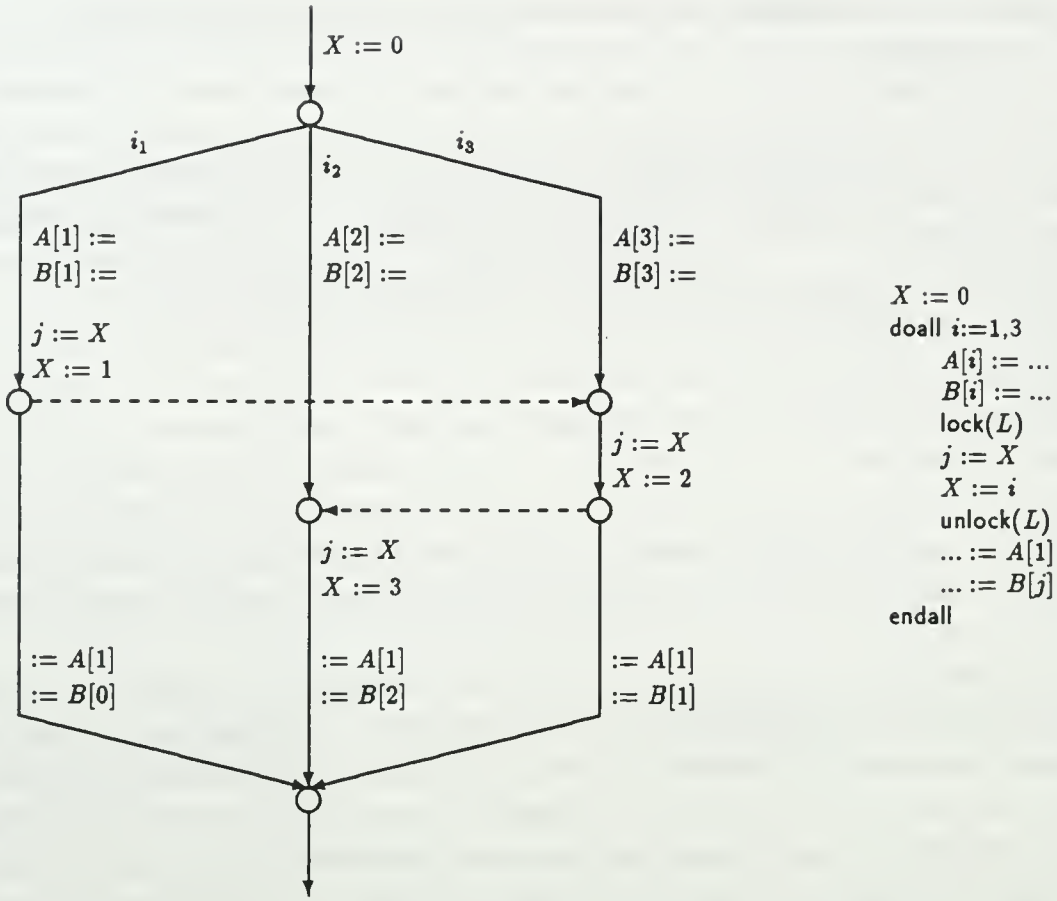


Figure 2.8: POEG with Three Thread Critical Section Coordination

- post(e) - The state of e becomes posted.
- clear(e) - The state of e becomes cleared.
- wait(e) - If the state of e is posted then the calling block proceeds. Otherwise it is suspended until the state of e becomes posted, at which point all waiting threads may proceed.

In contrast to message passing, no data is conveyed by an event. Therefore, events are used primarily for conveying condition statuses. Events have been implemented for several parallel Fortran systems such as Cedar Fortran [GPJL88].

The primary difficulty in representing event coordination stems from the fact that there is not a one-to-one correspondence between post and wait operations. Several post operations can precede a given wait operation. Emrath, Ghosh and Padua present a method for representing event synchronization on a *task graph* (roughly equivalent to a POEG) that is generated from an execution trace during post-mortem analysis [EGP89].

Each wait operation w has a *trigger set* that consists of the post operations that can possibly signal w. Post operations performed by block p cannot trigger a wait operation performed by block w if any of the following conditions hold:

1. Block p is a descendant of w.
2. Block p has a descendant block b such that b performs a clear or post operation and b is an ancestor of w.

The indirect parents of a wait operation w are those blocks that are ancestors of every block in the trigger set of w; these closest common ancestors are the blocks whose execution is guaranteed to proceed w. An iterative algorithm is given for finding the trigger set and its closest common ancestors. A similar technique is used in static analysis of parallel programs [CS88,CKS90].

To illustrate event coordination and trigger sets, consider Figure 2.9 which shows a POEG containing various post, clear and wait operations. The trigger set for the wait operation performed by block b_2 is $\{b_4, b_5\}$; it does not contain the post operation performed by b_7 (since b_7 is a descendant of b_2) or b_0 (since b_1 performs a clear operation and is a descendant of b_0 and an ancestor of b_2). The set of closest common ancestors of the trigger set $\{b_4, b_5\}$ is $\{b_3\}$. This means that an asynchronous coordination edge is added from block b_3 to block b_2 to capture the ordering resulting from the wait operation performed by b_2.

The on-the-fly computation of the trigger set and closest common ancestors[1] differs from the post-mortem computation in several ways:

1. The entire POEG is not available when coordination edges are added. In particular, when a wait operation is performed there may be post operations that should be included in its trigger set that have not yet executed. Therefore, the set of closest common ancestors may have to be calculated from a partial trigger set.
2. Whenever a clear operation is performed, the current trigger set is deleted. However, a clear operation does not affect future (and possibly concurrent) post operations.
3. The on-the-fly representation is not iterative; once an ancestor set is calculated it is not modified later during program execution.

The on-the-fly technique uses information about the actual order in which post and wait operations are performed during an execution instance. (The post-mortem approach, in

[1]The set of closest common ancestors is computed from the ancestor information used in task recycling (described in the Chapter 3) by creating an intersection parent vector whose i^{th} entry is the minimum of the i^{th} entries of the parent vectors of the blocks in the trigger set.

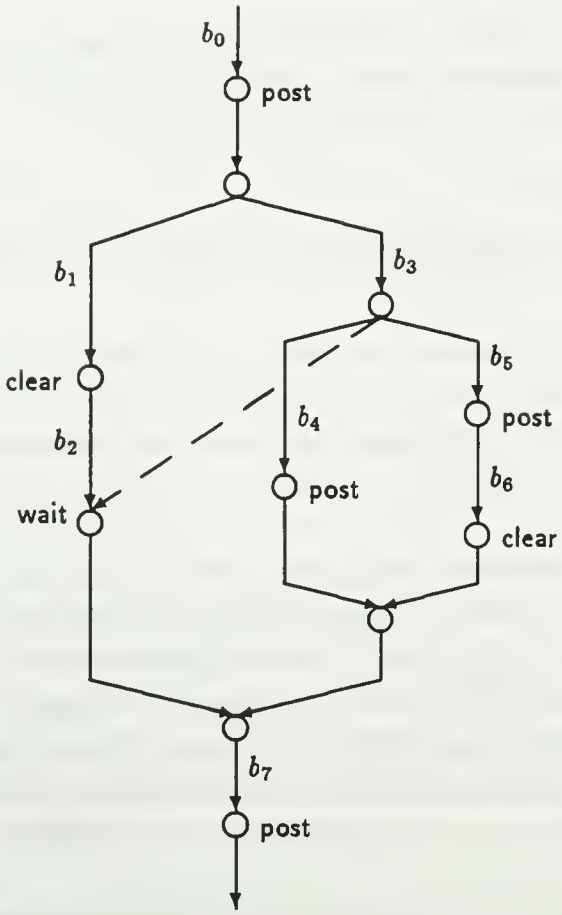


Figure 2.9: POEG with Event Coordination

contrast, attempts to compute a more global ordering relationship.) Because of this, the on-the-fly computation may add more ordering than intended by the programmer, and anomalies can be missed in a given execution instance. However, Theorem 3 proves that safe accesses will not be incorrectly reported as anomalies.

> **Theorem 3:** The on-the-fly representation for event coordination guarantees that two blocks that cannot execute concurrently will be ordered in the POEG.

This property does not hold for the post-mortem algorithm.

To illustrate the difference between the two techniques, consider the POEG in Figure 2.10. In the on-the-fly technique, there are several possible trigger sets of the wait operations performed by blocks b_1 and b_2, depending on the actual execution order of the post and wait operations:

1. If b_1 completes before b_2 then $T_{b_1} = \{b_6\}$ and an edge is added from b_6 to b_1.

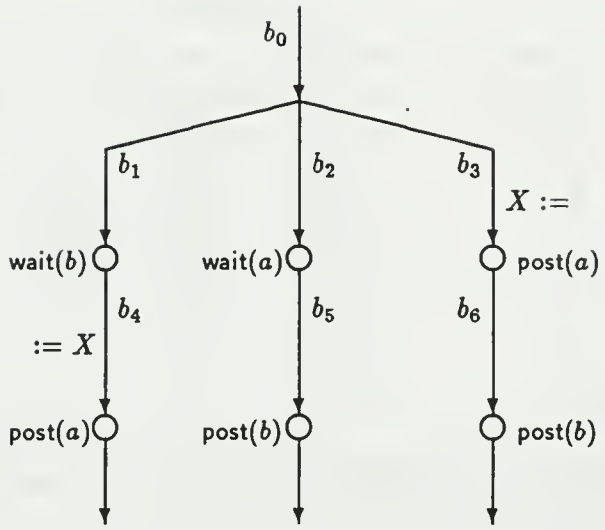


Figure 2.10: POEG with Event Coordination

There are several different trigger sets for b_2.

(a) $T_{b_2} = \{b_3\}$: An edge is added from b_3 to b_2.

(b) $T_{b_2} = \{b_3, b_4\}$: An edge is added from b_3 to b_2. (The edge from b_6 to b_1 has already been added making b_3 an ancestor of b_4.)

2. If b_2 completes before b_1 then $T_{b_2} = \{b_3\}$ and an edge is added from b_3 to b_2.

   There are several different trigger sets for b_1:

   (a) $T_{b_1} = \{b_6\}$: An edge is added from b_6 to b_1.

   (b) $T_{b_1} = \{b_5\}$: An edge is added from b_5 to b_1.

   (c) $T_{b_1} = \{b_5, b_6\}$: An edge is added from b_3 to b_1. (The edge from b_3 to b_2 has already been added making b_3 an ancestor of b_5.)

The three possible POEG's are shown in Figure 2.11. In all cases the read of X by b_4 and the write of X by b_3 are ordered, and hence no anomaly will be reported.

In contrast, the post-mortem algorithm adds a (redundant) coordination edge from block b_0 to block b_2 for the wait operation performed by block b_2 since b_0 is the closest common ancestor of blocks $\{b_3, b_4\}$. Likewise, the wait operation performed by block b_1 results in an edge from block b_0 to block b_1. (Block b_0 is also the closest common ancestor of $\{b_6, b_5\}$.) Therefore, the write to X performed by b_3 and the read of X by b_4 will be incorrectly identified as an anomaly.

Theorem 3 *The on-the-fly representation for event coordination guarantees that two blocks that cannot execute concurrently will be ordered in the POEG.*

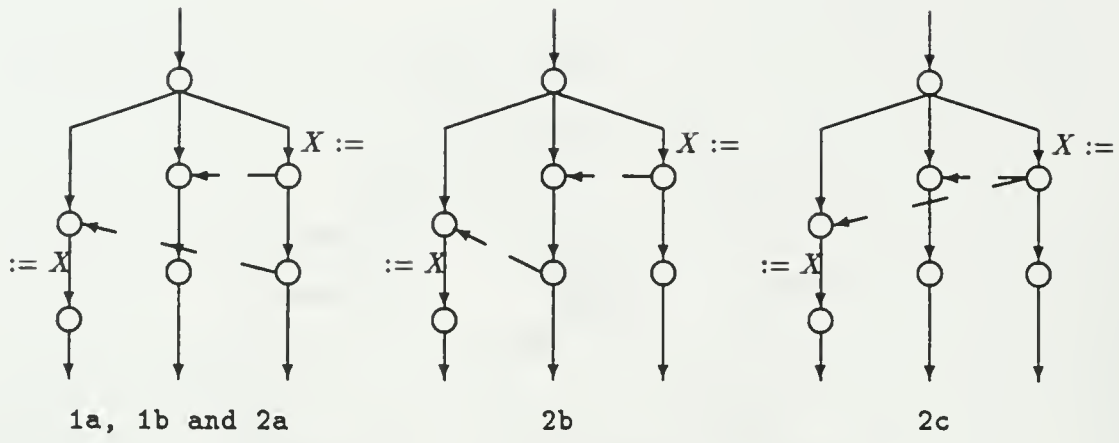


Figure 2.11: Three possible POEGs with Coordination Edges

Proof. Suppose, for the sake of contradiction, two blocks are incorrectly modeled as being concurrent in the on-the-fly representation of some execution instance E. Let a and w be the first blocks such that a must execute before wait operation w in all execution instances, but a and w are concurrent in E, and let T_w denote the trigger set of w.

Since w was eventually posted in E, we know that $T_w \neq \emptyset$. Block a must be concurrent with or a descendant of at least one block in T_w: if a were an ancestor of all blocks in T_w, a would be a common ancestor of T_w and therefore be ordered with w in E. Moreover, a must not be an ancestor of w.

Let E' be an execution instance that is identical to E except that the trigger set of w in E' consists of $T_w - T_a$ where T_a contains all members of T_w that are descendants of a. E' is a valid execution instance since no block in T_a is an ancestor of w, a or any member of $T_w - T_a$. (If a block in T_a was an ancestor of w then a would also be an ancestor of w.) Block a is either concurrent with or a descendant of every block that could possibly have triggered w in E'. Therefore, w can execute at the same time as a in E' and hence a contradiction has been reached. □

2.4 Reliability of Dynamic Access Anomaly Detection

The reliable detection of access anomalies is based on two properties:

Reliability Properties:

1. A program with no anomalies is not reported falsely as having anomalies.
2. A program with anomalies is reported as having anomalies.

It is not difficult to guarantee Reliability Property 1. To do so, a POEG representation must guarantee that two accesses that are ordered in every execution instance are not incorrectly reported as an access anomaly. All representations for the coordination primitives described in Section 2.3 meet Reliability Property 1. This can be seen easily for barrier, message passing, doacross, rendezvous, and critical section coordination; Theorem 3 proves it explicitly for event coordination.

However, guaranteeing Reliability Property 2 is more difficult. The primary weakness of the dynamic access anomaly detection approach is that it only conveys information about a single execution instance of a program; therefore, it is a priori limited to proving properties about a given input vector[2]. The problem of detecting access anomalies in a global sense for a program can be addressed only by static analysis techniques.

Even when the input vector is fixed, the absence of access anomalies in one execution instance of a program does not necessarily indicate that the program will never have an access anomaly. We say that a program and input vector pair (P, I) is *anomaly free* if and only if no execution instance of P on I contains an access anomaly. If we are unable to prove that a program and input vector pair is anomaly free, we can have little confidence in the correctness of the program. The difficulty of guaranteeing freedom from anomalies lies in the number of execution instances that must be analyzed.

Once an input vector is defined, there are only two sources of nondeterminism in a parallel program. One source of nondeterminism stems from access anomalies which can modify the subsequent execution of the program. Since our goal is to guarantee that the program and input vector pair is anomaly free, detecting the first occurrence of an anomaly is sufficient. The second source of nondeterminism stems from certain coordination operations; for example, critical section coordination. Thus, even in the absence of access anomalies there can be many different execution instances for a given program and input vector pair. While programs would be easier to write and debug if this nondeterminism were outlawed, nondeterminism is in fact useful for a wide variety of applications and thus, necessary.

We distinguish between three types of nondeterministic behavior that can be found in parallel programs: internal and external nondeterminism (defined by Emrath and Padua [EP88]) and POEG nondeterminism.

Definitions:

Internal Determinism : The sequence of instructions performed by each thread, as well as the variables used by those instructions, is deterministic.

[2]An input vector contains *all* external input to the program including, for example, values returned from a random number generator, timing routines, or interrupts.

External Determinism: The output of the program is deterministic, but the program in internally nondeterministic.

POEG Determinism: The structure of the POEG is deterministic.

Most parallel programs are externally deterministic.

There is a subtle relationship between internal nondeterminism and POEG nondeterminism. By definition and construction, all nondeterminism resulting from coordination operations is explicitly modeled in the POEG: the only difference in two execution instances of a program and input vector pair that map to the same POEG derives from access anomalies. In the absence of access anomalies, each individual block in a POEG is internally deterministic. If the nondeterminism arising from varying interthread coordination were not represented in the POEG, then the program would be POEG deterministic, but each block could be internally nondeterministic.

In software testing theory, [GG77,WO80], a set of inputs T is chosen to satisfy a *data selection criterion* C where C denotes some predicate over subsets of the domain D of all possible inputs. A selection criterion is *reliable* if and only if the program succeeds or fails consistently when executing a test input that satisfies C. A selection criterion is *valid* if and only if there exists a set of inputs satisfying C that will not be executed correctly if the program is incorrect. A set of test inputs T is *successful* if the program succeeds on each member of T. The fundamental theorem of testing is based on these definitions:

$$(\exists_C)(valid(C) \wedge reliable(C) \wedge (\exists_T \subseteq D)(C(T) \wedge successful(T)) \supset successful(D)$$

In other words, a successfully executed test suite is equivalent to a direct proof of correctness if the test suite satisfies a data selection criterion proved to be both valid and reliable.

In the context of dynamic access anomaly detection, the domain of test inputs is the entire collection of possible execution instances for a given program and input vector pairs (since we are interested in proving properties about a given input vector). The first result about the complexity of guaranteeing Reliability Property 2 is Theorem 4.

> **Theorem 4:** A valid and reliable data selection criterion for guaranteeing that a program and input vector pair (P, I) is anomaly free is a set of execution instances whose POEGs cover the entire range of graphs possible for P and I.

It follows from Theorem 4 that the number of execution instances that must be analyzed to guarantee that a program and input vector pair (P, I) is anomaly free is bounded by the number of different POEGs for P and I. Given some tool for generating execution instances, theoretically we can prove that a program is anomaly free for a given input vector.

Unfortunately, the number graphs that exist for POEG nondeterministic programs makes even this task intractable. Let N_i denote the number of different execution orders of the coordination operations for each initial program state of a set of concurrent threads C_i. (The *program state* at any point in the program execution is the current value of all variables that are later referenced.) Each execution order of C_i can generate a new program state at the beginning of the subsequent execution. Thus, a program that consists of two parallel constructs, C_1 followed by C_2, has $N_1 \times N_2$ possible POEGs. In general, the number of different graphs for a program with POEG nondeterminism is a product of the number of possible sources of nondeterminism.

Fortunately, not all programs are POEG nondeterministic. A direct consequence of Theorem 4 is Corollary 1.

> **Corollary 1:** One execution instance is sufficient to guarantee that a POEG deterministic program and input vector pair is anomaly free.

To apply Corollary 1, one must be able to classify programs as POEG deterministic and POEG nondeterministic.

POEG nondeterminism arises when race conditions in the execution order of coordination operations affects the structure of the POEG. For example, the order in which a set of threads reach a barrier does not affect the representation in the POEG. However, the actual order in which two threads execute a critical section is explicitly represented in the POEG. Therefore, a simple way to detect if a program is POEG nondeterministic is to consider the types of coordination primitives that it contains.

We can distinguish between two types of coordination primitives:

Definition:

> A coordination primitive is *matched* if and only if the edges required to represent the coordination between b_i and b_j in any POEG are identical in all execution instances in which the initial program states of b_i and b_j are identical.

Otherwise, the coordination primitive is *unmatched.* The concept of matched coordination provides a method for identifying POEG deterministic programs.

> **Theorem 5:** Every program whose only parallel operations are fork and join operations and matched coordination is POEG deterministic.

> **Theorem 6:** The barrier coordination primitive, message passing coordination with explicitly named senders and receivers, and doacross coordination primitive are matched.

Therefore, parallel programs that only contain certain parallel operations (namely, fork and join operations and barriers, named message passing, and doacross coordination) can

be guaranteed to be anomaly free for a given input vector simply by examining a single execution instance. This subclass is important since it includes many parallel scientific programs.

Many common types of coordination–including Ada rendezvous, critical section coordination and events–are unmatched. In general, any type of *guarded* or *unnamed* coordination is unmatched. (If a coordination operation is guarded, then whether or not it is performed depends on the state of other threads [Dij75]; coordination is unnamed if the pair of coordination threads is not predefined.) However, not all programs that contain unmatched coordination are POEG nondeterministic. For example, the execution of many programs with critical section coordination is independent of the execution order of the critical sections. Chapters 5 considers the specific types of nondeterminism that can arise from critical section coordination and discusses techniques for determining when nondeterminism is present and how it affects the detection of access anomalies.

Identifying a program as POEG deterministic simplifies the analysis of the program in several additional ways:

1. Once a POEG deterministic program has been proven to be anomaly free, the complexity of showing other properties of the program is greatly simplified. For instance, since the structure of the POEG is deterministic, a single execution instance is sufficient to guarantee that the program and input vector pair will never deadlock.

2. There is no probe affect when detecting access anomalies for POEG deterministic programs. The structure of the POEG–and hence the concurrency relationship–is unaffected by changes in the program behavior due to the insertion of monitoring code.

3. POEG determinism simplifies the task of trace-and-replay debugging of parallel programs (for an example see the work of Fowler, LeBlanc and Mellor-Crummey [LMC87,FLMC88]). In particular, matched coordination operations do not need to be traced since the same set of coordination operations will occur in every execution instance. If a program is POEG deterministic then no coordination operations need to be traced or replayed.

4. Testing POEG deterministic programs with no access anomalies is relatively simple. The formal model for testing parallel programs proposed by Weiss [Wei88] is based on *sequentializations* of a parallel program. In an anomaly free program, the POEG captures all interblock interaction. Thus, one sequentialization is sufficient to test an anomaly free POEG deterministic parallel program for a given input vector.

This gives one hope that dynamic anomaly detection can be combined with testing methodologies for generating input vectors and other debugging techniques to guarantee that a POEG deterministic program is correct as well as anomaly free.

Lemma 1 *In the absence of access anomalies, the initial program state of all blocks are identical in all execution instances of a program and input vector pair that are represented by the same POEG.*

Proof. Proof by induction on the length L of the longest path from the root of the POEG to a block b.
Base Case ($L = 0$). The program state of the root block is deterministic since it is the initial input vector.
Induction Hypothesis: Assume that the lemma holds for all blocks with $L < i$. Prove that for all blocks b with $L = i$ the initial program state of b is identical in all execution instances E_i that are represented by the same POEG.

Since all execution instances E_i are represented by the same POEG, b must have the same set of parent blocks in all E_i. The initial program state of all parent blocks of b are identical in all E_i by the induction hypothesis (the longest path from the root to a parent of b is guaranteed to be less than i). Since there are no access anomalies, every block performs the same sequence of actions in every execution instance and no shared variable is written by two parent blocks. Therefore, the final program state of each parent block is identical and non-conflicting in all execution instances. The initial program state of b is simply a union of the final program states of its parent blocks and hence must be the same in all execution instances. □

Theorem 4 *A valid and reliable data selection criterion for guaranteeing that a program and input vector pair (*P, I*) is anomaly free is a set of execution instances whose POEGs cover the entire range of graphs possible for* P *and* I.

Proof. Suppose, for the sake of contradiction, that some execution instance E has an access anomaly that is not detected. E must have the same POEG as some execution instance E' that does not have any access anomalies since all possible POEGs for the program and input vector pair were analyzed.

In order for there to be an anomaly in E and not in E', the set of variables read and written by some block must be different in E and E'. Consider the set of concurrent blocks B that contain access anomalies in E such that no ancestor of a block in B contains access anomalies. By Lemma 1, we know the initial program states of the blocks in B are the same in E and E'. Since all sources of nondeterminism are fixed for all blocks $b \in B$, the actions performed by each b are identical before the first anomaly.

We can create a partial order of the accesses in B by linearizing: (i) the accesses in each block, and (ii) the accesses to each variable. Let a be an access in E that has a different value in E' such that all accesses ordered before a are identical in E and E'. If a is a definition, it is a function of a set of uses. Since these uses are ordered before a by rule (i), they must have the same values in both execution instances. Thus, the value computed for a also must be the same. If a is a use, it is reached by some definition, d. Since d is ordered before a by rule (ii), d must be identical in both execution instances. Thus, the value that reaches a must also be the same. Hence, every access must be identical in E and E' and therefore an anomaly is detected in E if and only if an anomaly is detected in E'. □

Corollary 1 *One execution instance is sufficient to guarantee that a POEG deterministic program and input vector pair is anomaly free.*

Proof. Follows directly from the definition of POEG determinism and Theorem 4. □

Theorem 5 *Every program whose only parallel operations are fork and join operations and matched coordination is POEG deterministic.*

Proof. Proof by induction on the length L of the longest path from the root to the block.
Base Case ($L = 0$) The execution of the root block is deterministic and anomaly free.
Induction Hypothesis: Assume that the theorem holds for all blocks with $L < i$. Prove that for blocks b with $L = i$ that the structure of the POEG is deterministic up to and including the creation of b.

By the induction hypothesis, the structure of the POEG up to and including the creation of the parent blocks of b is deterministic. Therefore, by Theorem 4 the initial program state of the parent blocks of b–and hence the actions performed by the parent blocks–are deterministic. Consider all possible ways in which b could be created:

- Fork: In the absence of access anomalies, each fork operation is a simple instruction whose execution is independent of the actions of concurrent threads. Thus, the parent of b creates the same children blocks in all execution instances.
- Join: Each join operation is a simple instruction based on a preceding fork operation. Thus, the parents of b all perform the same join operation in every execution instances.
- Matched Coordination: By Lemma 1 the initial program states of the parent blocks of b must be identical in every execution instance. Since all coordination is matched, by the definition of matched coordination, the execution of parents of b results in the same set of coordination edges in all execution instances.

In all cases the structure of the POEG is deterministic. □

Theorem 6 *The barrier coordination primitive, message passing coordination with explicitly named senders and receivers, and doacross coordination primitive are matched.*

Proof. We will consider each coordination primitive separately.

> **Barrier:** By definition, it is semantically incorrect if either only $n-1$ or $n+1$ concurrent threads to perform a given n-way barrier operation[3]. Since the execution of the program up to the point of the barrier is deterministic, the only way to enforce this correctness constraint is if the set of threads executing a barrier operation is fixed a priori.
>
> **Message Passing:** In the absence of access anomalies and preceding POEG nondeterminism, the execution of each block is deterministic and hence each block always executes the same series of named send and receive operations. In particular, the receiver will always receive the messages in the same order regardless of the order they were actually sent. Therefore, the same set of coordinations will occur in every execution instance of a given program and input vector pair.
>
> **Doacross:** This follows directly from the above argument since doacross coordination is a restricted form of named message passing.

In all cases the structure of the POEG is deterministic and hence the coordination primitives are matched. □

[3]This can be detected easily during anomaly detection.

Chapter 3

Task Recycling Technique

This chapter presents the task recycling technique for detecting access anomalies during program execution. Task recycling improves upon existing dynamic access anomaly detection techniques in three significant ways:

1. *Generality.* The task recycling technique is applicable to a wide variety of thread creation and termination primitives and all common synchronous and asynchronous coordination primitives. This is an improvement over the methods of Hood, Kennedy and Mellor-Crummey [HKMC89], Choi, Miller and Netzer [Cho89,CMN88,MC88] and Nudler and Rudolph [NR88a,NR88b] which only support a subset of the common coordination and thread creation primitives.

   Task recycling has been used to find anomalies in programs written in a parallel Fortran dialect by Dinning and Schonberg [DS90] as well as in Ada by Hind, Ishbiah, Konany and Schonberg [SS86,HIKS]. Chapter 4 presents empirical measurements of the Fortran implementation.

2. *Performance.* Monitoring costs are incurred at thread create, terminate, and coordinate operations, and every time a monitored variable is accessed. Because variable accesses are generally the most frequent operation, the task recycling technique is preferred, since it reduces the overhead per variable access to a small constant. While the worst case cost at parallel operation is linear in the maximum concurrency of the graph, the actual cost depends on the complexity of synchronization patterns and on the nesting structure of parallel constructs. Experiments show that the parallel structure of a large class of scientific programs is very simple. For these programs, the expense incurred at parallel operations can be significantly reduced (as low as O(1)) by using standard types of data structure transformations.

3. *Storage.* The task recycling technique derives its name from the fact that it reuses task identifiers. This enables task identifiers to be simple integers, so that the

storage cost for concurrency information depends only on the maximum concurrency of the POEG. In contrast, the number of task identifiers in English-Hebrew labeling [NR88a,NR88b] grows with the length of execution, and each identifier is a complex object making storage management an issue.

To detect anomalies, the task recycling technique must first determine which variables are accessed by each block. Section 3.1 describes how this information is maintained in *access histories* associated with monitored variables. The access history of a shared variable X stores the identifiers of the blocks that most recently accessed X. When a new block accesses X, the access history of X is compared with the information about the block's concurrency relationship to check whether there is an access anomaly involving X.

Secondly, we must be able to determine which blocks can potentially execute concurrently by analyzing and encoding the POEG. The ancestor relationships is stored in *parent vectors*, as described in Section 3.2. A parent vector summarizes the thread creation, termination, and coordination history in a compact manner that reduces the cost of checking if two blocks are concurrent to a small constant.

The problem of assigning *tasks identifier* labels to blocks is discussed in Sections 3.3 and 3.4. The goal of a task assignment algorithm is to minimize the number of tasks used in assigning blocks by recycling tasks: the number of tasks determines the length of parent vectors.

The techniques developed for detecting access anomalies can be integrated with other parallel debugging tools to increase their functionality and robustness. Section 3.5 discusses extensions to two of the most common methods for debugging shared memory parallel programs: trace-and-replay debuggers and event-based monitoring. For the remainder of this chapter, the symbols in Table 3.1 are used to denote various parameters.

Symbol	*Description*
A	the number of tasks used in a task assignment
B	the number of blocks in the POEG
N	the maximum level of nesting of fork-join constructs
T	the maximum concurrency of the POEG
T'	the maximum concurrency of the POEG if coordination edges are ignored
V	the number of monitored variables
W	the number of blocks created by a fork operation

Table 3.1: Table of Symbols

3.1 Maintaining Access Histories

One of the primary drawbacks of trace-based anomaly detection is the potentially very large storage requirements. The amount of variable access data is proportional to the length of the program execution times the maximum concurrency of the program. In on-the-fly detection of anomalies, information that is no longer needed can be discarded as the program executes. In particular, task recycling compresses information about variable accesses so that storage is bounded by $O(V \times T)$. By reducing the amount of data stored, the work required to check each variable access is also decreased.

Although compaction of events reduces the amount of variable access data, certain information is lost. In particular, if there are multiple anomalies involving the same variable some of them may not be reported. The primary requirement in on-the-fly anomaly detection is that enough information is saved to guarantee that at least one anomaly is reported for every variable which is accessed in an "unsafe" manner. This level of error reporting is generally considered to be sufficient, since any execution after the first access anomaly is suspect. A secondary goal is to maintain enough information so that the sequence of accesses to a shared variable can be reproduced.

To evaluate Bernstein's conditions, the task recycling technique stores Read and Write sets in an inverse format by associating information with each monitored variable. The *access history* of a variable X contains information about the blocks which have read and written X. Although we use the term "variable", access histories are actually associated with virtual addresses. Therefore, variable aliasing–one of the primary weaknesses of static anomaly detection–is not an issue. Access anomalies can be accurately detected even in programs with pointer and complex array indexing.

Every time variable X is read or written, its access history is examined to determine whether the current read or write event conflicts with a previous one. In particular, when block b reads X, block b checks to see whether or not it is concurrent with the writers in the access history for X. When block b writes X, b checks if it is concurrent with any of the blocks in the access history of X. If either condition is true, an access anomaly is reported. Block b then adds its block label to the appropriate set in the access history. The algorithms for checking for conflicts at read and write events are shown in Algorithm 3.1.

The efficiency of detecting anomalies primarily depends on: (1) how quickly we can determine if two blocks are concurrent, and (2) how many entries are in an access history. Therefore, a goal of on-the-fly anomaly detection is to minimize the size of access histories. (Section 3.2 discusses an efficient concurrency check.) The notion of storing access information with monitored variables was first proposed by Nudler and Rudolph [NR88a,NR88b] who save only the most recent reader and the most recent writer. This

```
procedure Check-Read(b, X)
    for all a ∈ Writer(X) do
        if IsConcurrent(b,a) then report Access Anomaly
    end for
end procedure

procedure Check-Write(b, X)
    for all a ∈ Reader(X) ∪ Writer(W) do
        if IsConcurrent(b,a) then report Access Anomaly
    end for
end procedure
```

Algorithm 3.1: Check for Read and Write Events

data is insufficient to guarantee that every variable that is accessed in an unsafe manner is detected. If all information about past accesses is maintained, however, the size of the access histories will grow to be equal to the size of the trace data. We would like to store only that information required to determine if a variable is being accessed in an unsafe manner and delete all redundant information.

Consider the program in Figure 3.1 and suppose that a variable X is read by blocks b_1, b_2, and b_3 and written by block b_4 in that order. The write of X by b_4 conflicts with the reads of X by b_1 and b_2. After the first read, the access history of X contains b_1. When X is next read by b_2, block b_2 is added to the access history and b_1 is deleted. Once b_2 is added to the reader set, the b_1 read event is no longer needed: any subsequent write event that conflicts with b_1 also conflicts with b_2. On the other hand, when b_3 reads X and is added to the access history, the b_2 read event cannot be deleted. Otherwise, no anomaly can be detected when X is written by b_4, since this write conflicts with the read by b_2 but does not conflict with the read by b_3.

More generally, a block b in the reader (or writer) set of the access history of a variable X can be *subtracted* from the access history when a descendant of b reads (or writes) X. Thus, a set in the access history contains two blocks only if they are concurrent. In addition, an entry in an access history can be subtracted if it conflicts with the current access[1]. By deleting obsolete entries, the size of the reader set is bounded by the maximum concurrency of the program, T. Since two concurrent writes always conflict, there is at most one writer in an access history. Algorithm 3.2 shows the subtraction algorithms for read and write events.

The general subtraction rule is that an event a is subtracted by an event e only if

[1] In order to maintain enough information to attain the secondary goal of reproducibility, a reader does not delete a concurrent writer from an access history. This deletion can be performed if the only goal is to detect at least one anomaly for every variable that is accessed in an unsafe manner.

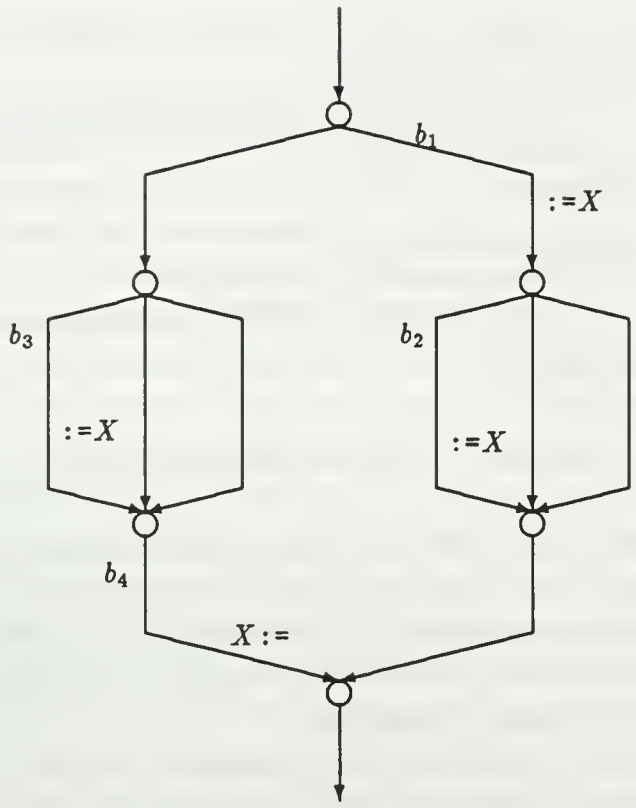


Figure 3.1: Access History Example

every future event that conflicts with a also conflicts with e. Therefore, every variable X ...at is accessed in an unsafe manner is guaranteed to be detected. In addition, enough access data is maintained to achieve the second goal of on-the-fly detection: specifically, to provide enough information about the execution order of accesses so that the sequence of accesses to X can be reproduced. (This is proved in Section 3.5.)

The bound on the size of the reader set of T is much too pessimistic. Chapter 4 shows that for typical numeric parallel Fortran programs, the average access history size is very small and independent of T. This is due to the common practice of data partitioning to optimize performance. For many programs a reader set with two entries is sufficient for most variables and thus the total space for access histories is $O(V)$. In fact, for programs with no nested parallelism or coordination, a reader set of size two is always sufficient to detect all variables that are accessed in an unsafe manner. However, not enough information is maintained to reproduce a sequence of accesses to a variable.

```
procedure Subtract-Read(b, X)
    for all a ∈ Reader(X) do
        if not IsConcurrent(b,a) then
            Reader(X) := Reader(X) − a
        end if
    end for
    Reader(X) := Reader(X) + b
end procedure

procedure Subtract-Write(b, X)
    for all a ∈ Reader(X) do
        if not IsConcurrent(b,a) then
            Reader(X) := Reader(X) − a
        end if
    end for
    Writer(X) := b
end procedure
```

Algorithm 3.2: Subtraction for Read and Write Events

3.2 Detecting Concurrency

Concurrency information is used at every variable access and updated at every parallel operation. Because we expect accesses to monitored variables to be more frequent than parallel operations, the task recycling technique is designed to make the cost of monitoring a variable access as small as possible. To do so, the data structure for maintaining ancestor information minimizes the cost of concurrency checking. There is a small constant amount of work required to determine if two blocks are concurrent, since this check must be performed for every entry in the access history of variable X every time that X is accessed. However, the cost at fork, join, and coordination operations can be relatively high: in the worst case it is proportional to the maximum concurrency of the POEG, T.

In order to bound the space required for maintaining concurrency information, the task recycling algorithm never explicitly builds or analyzes an entire POEG during execution. Instead, the concurrency relationship inherent in the structure of the POEG is computed incrementally as the program executes and stored in a compact format. Because we are monitoring during program execution, a block whose tag is in an access history that is not an ancestor of an executing block b must be concurrent with b. Thus, only information about the ancestor relationship is maintained.

The ancestor relationship is encoded by a unique *task identifier* label t_v associated with each block that consists of a *task* t and a *version number* v. More than one block may be assigned to the same task at different times in the program execution. The rule

for recycling tasks is that once a block completes, its task can be reassigned to any of its descendant blocks. Therefore, two blocks that are successively assigned the same task must be ordered by the ancestor relationship. The version number of a task identifier is used to distinguish among different blocks assigned to the same task; every time a block is assigned to a task t, the associated version number v is incremented. A block with task identifier t_i is an ancestor of a block with task identifier t_j if and only if $i < j$.

Figure 3.2 shows a valid task assignment of a partial POEG. This assignment is valid

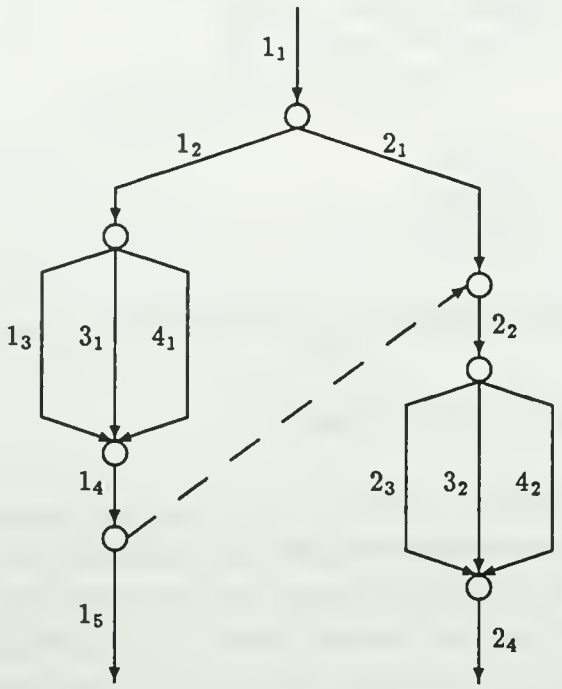


Figure 3.2: Valid Task Assignment

since no task is assigned to concurrent blocks. For example, task 3 is assigned to two blocks that are ordered by a coordination edge. When task 3 was reassigned its version number was incremented.

Task identifiers contain very little information about the ancestor relationship. Additional concurrency information is maintained in a *parent vector* associated with each currently executing block. Data structures similar to parent vectors have been used before. For example, parent vectors correspond to the time vectors used by Fidge [Fid88], and the *before* vectors in the post-mortem analysis technique of Choi, Miller and Netzer [Cho89,CMN88,MC88][2].

There is an entry in the parent vector associated with each task t that contains in-

[2]Because we monitor on-line, only a limited number of parent vectors exist at any point in the execution.

formation about the ancestors that were assigned task t. Specifically, the t^{th} entry in the parent vector for block b stores the version number of the ancestor of b that was most recently assigned task t (e.g. the ancestor with the largest version number). An executing block b can tell if it is concurrent with a block assigned task identifier t_v by comparing v to the t^{th} entry in the parent vector of b. Algorithm 3.3 shows the code for determining if block b is concurrent with task identifier t_v.

```
procedure IsConcurrent(b, t_v)
    if parent(b)[t] < v then
        return true
    else return false
end procedure
```

Algorithm 3.3: Check Concurrent

New parent vectors are formed from the vectors associated with parent blocks as in [Cho89,CMN88,MC88]. When one of the parents of a new block c has task identifier t_v, the t^{th} entry of the parent vector of c is v. Otherwise, the t^{th} entry of the parent vector of c is the maximum of the t^{th} entry of the parent vectors of the parents of c. Thus, when blocks $p_1 \ldots p_m$ with task identifiers $t_{1v_1} \ldots t_{mv_m}$ create a new block c, the parent vector of c is initialized using Algorithm 3.4.

```
procedure Build-Parent-Vector(c, p_1 ... p_m)
    for i := 1 to A do
        if there exists p_j ∈ {p_1 ... p_m} such that i = t_j then
            parent(c)[i] := v_j
        else parent(c)[i] := max(parent(p_1)[i], ..., parent(p_m)[i])
    endfor
end procedure
```

Algorithm 3.4: Build Parent Vector

To illustrate the use of parent vectors, consider Table 3.2 which shows the parent vectors for the POEG in Figure 3.2. (While parent vectors for the entire graph are shown, only those for the blocks currently executing, 1_5 and 2_4, exist at one time.) Two ancestors of block 2_4 were assigned task 3–namely 3_1 and 3_2. Therefore, the third entry of the parent vector of block 2_4 contains 2, since this is the version number of the ancestor of 2_4 that was most recently assigned task 3. The parent vector of block 2_2–the child of the coordination edge–contains 2_1 and 1_4 as well as all of the ancestors of 1_4. The parent vector for 2_4 is calculated as the entry-wise maximum of the parent vectors of its parent blocks–2_3, 3_2, and 4_2–and their task identifiers. Lastly, we can tell that block 1_5 is

Task Ids	*Parent Vector*
1_1	[0,0,0,0]
$1_2, 2_1$	[1,0,0,0]
$1_3, 3_1, 4_1$	[2,0,0,0]
1_4	[3,0,1,1]
2_2	[4,1,1,1]
$2_3, 3_2, 4_2$	[4,2,1,1]
1_5	[4,0,1,1]
2_4	[4,3,2,2]

Table 3.2: Parent Vector Table

concurrent with block 2_1 since the second entry in the parent vector of 1_5, [4,0,1,1], is less than 1. However, block 1_5 is not concurrent with 3_1 since the second entry in the parent vector of 1_5 is not less than 1.

The primary benefits of using task identifiers and parent vectors for encoding the concurrency relationship of the POEG are two-fold. First, there is a small constant cost for determining whether the executing block is concurrent with another block. In particular, the cost of telling if two blocks are concurrent is an array index and a comparison operation. Second, only a small amount of information is needed about a previous block for a currently executing block to determine if they are concurrent. Storing a task identifier in an access history is sufficient to perform the concurrency check at a variable access. Parent vectors are necessary only for the blocks that are currently executing; parent vectors of terminated blocks are discarded. This is an improvement over English-Hebrew labeling in which a complex object (string of integers) is associated with each entry in an access history, and trace-based methods where ancestor and descendant information for all blocks is maintained.

The primary drawback of task recycling is the cost associated with maintaining parent vectors. The length of each parent vector is equal to the number of tasks A. (Section 3.4 shows that A is in the worst case bounded by T', the maximum concurrency of the POEG if coordination edges were removed.) Because of the coordination constraint, the total amount of work required for parent vector maintenance is bounded by $O(A)$ per block. The number of parent vectors associated with executing blocks is no more than the maximum concurrency so that the total space for these parent vectors is $O(T \times A)$. In addition, parent vectors are associated with outstanding asynchronous coordination operations.

One way to decrease the overhead associated with parent vector maintenance is by performing a good task assignment; e.g. a task assignment that minimizes the number

of tasks used. Issues in performing good task assignments are discussed in Sections 3.3 and 3.4. A second way to decrease the costs is to take advantage of the very simple concurrency structure of many parallel programs.

Data structure transformation techniques can be used to limit the work and space requirements of maintaining parent vectors for program that have simple concurrency structures. For example, parallel Fortran programs seldom have nested parallelism or coordination so that many concurrent blocks have the same or similar ancestor sets. By performing data structure transformations, the worst case overheads associated with parent vector maintenance are incurred only in programs with complex coordination patterns. One strength of the task recycling method is that its performance degrades gracefully when these transformations are used; threads that coordinate pay an extra cost, but threads that do not coordinate do not perform any additional work.

One obvious optimization is that two blocks with the same set of ancestors can share a parent vector. (If local memory is available and the cost of accessing shared memory is much higher than local memory, sharing of data structures may not be cost-effective.) For example, all blocks created by a doall operation have the same ancestors; a block must obtain a private parent vector only if it terminates before the endall (by performing a nested parallel construct or coordination operation). For a parallel construct with parallelism of W, the number of parent vectors is reduced by a factor of W. Thus, for a program with no nested parallelism or coordination, only a single parent vector is maintained.

In addition, by a proper choice of data structure, the parent vector computation overhead for innermost parallel constructs with no coordination can be reduced to an average cost of O(1) per block. Let *parent* be the shared parent vector of the blocks executing after an innermost fork operation. At the corresponding join operation, rather than creating a new parent vector for the block j executing after the join operation, *parent* is updated and reused for j. When a block with task identifier t_v terminates in the join operation, it must update the t^{th} entry of *parent* to be v. However, this update cannot be performed until all of the blocks that share *parent* have reached the join operation. Instead, the identifiers are buffered until all children complete and is reflected in *parent* by block j immediately before j begins execution.

The average cost of maintaining parent vectors can also be reduced in the presence of coordination. For instance, consider an innermost parallel construct C such that no thread created in C coordinates with a thread outside of C. The amount of work required for the fork and join operations is still O(1). *Modification-sets* can be used to bound the cost of updating a parent vector at a coordination point to be a function of the number of indirect parents of the coordinating block.

A modification-set is associated with every thread and stores the indirect parents of the currently executing block. The modification-set of the first block b in a thread is initialized with the task identifier t_v assigned to b. The first time a thread coordinates, a private parent vector is created and initialized. The modification set and parent vector for block b are updated after a coordination point with indirect parents $p_1 \ldots p_m$ by executing the code shown in Algorithm 3.5. In addition, if two task identifiers in the modification set have the same task, then the one with the smaller version number is deleted.

```
procedure Build-Parent-Vector(b, p1 ... pm)
    if first coordination then
        parent(b) := copy(parent(b))
    endif
    for all p ∈ pi ... pm do
        for all tv ∈ modification-set(p)
            parent(b)[t] := v
        end for
        modification-set(b) := modification-set(b) ∪ modification-set(p)
    end for
    modification-set(b) := modification-set(b) + tb,vb
end procedure
```

Algorithm 3.5: Build Parent Vector Using Modification Set

The cost incurred at a coordination point increases with the number of indirect ancestors which is a function of the pattern of coordination. $O(N)$ modification-sets can be used to support more general coordination patterns.

Lastly, one can consider optimizations that work only for "important" subclasses of parallel programs. For instance, the class of parallel programs supported by the on-the-fly technique of Hood, Kennedy and Mellor-Crummey [HKMC89] contains programs with parallel constructs and doacross coordination with distance 1. This anomaly detection technique requires $O(1)$ work for managing ancestor information at parallel operations and $O(log(N))$ work to determine if two blocks are concurrent.

The costs incurred by task recycling can be similarly decreased for this class of parallel programs. Specifically, the cost of maintaining parent vectors at doacross coordination points can be eliminated by associating a shared *iterate* vector with the shared *parent* vector. The t^{th} entry of *iterate* contains the iterate of the loop that is assigned to task t, if any. Otherwise, the t^{th} entry contains $W + 1$ where W is the number of iterates created. A block b is in a higher iterate than a block with task identifier t_v if and only if the iterate number of b is higher than $iterate[t]$. The *iterate* vector can be used to determine if two blocks are concurrent after a sequence of doacross coordinations, without storing any additional parent information.

After every doacross operation, each block increments its version number. The original parent vector check is used to tell if a block b is concurrent with a block with task identifier t_v whenever t_v is in a higher iterate than b. If t_v is in a lower iterate than b, block b first normalizes the version number stored in the parent vector by the number of doacross operations that b has executed, and then performs the standard parent vector check. Algorithm 3.6 shows the algorithm used to perform the check where k is the number of doacross operations that have been executed. This optimization only slightly increases the cost of telling if two blocks are concurrent while eliminating all parent vector maintenance at coordination points.

```
procedure IsConcurrent(b, t_v)
    if my-iterate < iterate[t] then
        if parent[t] < v then
            return true
        else return false
    else
        if parent[t] + k < v then
            return true
        else return false
    endif
end procedure
```

Algorithm 3.6: Check Concurrent Using Iterate Vectors

Other optimizations are discussed in Chapter 4.

3.3 Task Assignment Problem

The use of parent vectors to store the ancestor relationship assumes the existence of an algorithm for assigning tasks to blocks. A block b can be validly assigned to a task t if and only if b is not concurrent with any other block previously assigned to t. Since the number of tasks used in a task assignment determines the length of parent vectors, the goal of a task assignment algorithm is to use as few tasks as possible while maintaining a valid task assignment. Theorem 7 gives the lower bound on the number of tasks in a valid task assignment.

> **Theorem 7:** The minimum number of tasks needed to perform a valid assignment to a POEG is equal to the maximum concurrency of the POEG.

Any valid task assignment that uses the minimal number of tasks is said to be *optimal* [3]. This and the following section discuss the task assignment problem and present several

[3] This property was first proved by Dilworth and defined as the number of chains required to cover every element in a general partial order [Dil50,Per63].

task assignment algorithms.

There are several different systems that use data structures similar to parent vectors for maintaining concurrency relationships [Fid88,MC88]. All of these systems assume a constant degree of parallelism throughout the program execution, thereby avoiding the issue of task assignment. (Since every block has exactly one direct parent and one direct child, each block is simply assigned the same task as its direct parent block with an incremented version number.) The following algorithms can also be used to expand the class of parallel programs supported by these systems.

Let us first consider *off-line* task assignment in which the entire POEG is available when the assignment is performed. Theorem 8 gives an upper bound on the work required to perform an optimal assignment.

> **Theorem 8:** The complexity of performing an off-line optimal task assignment is equivalent to the complexity of maximum bipartite matching.

> **Corollary 2:** There exists an $O(B^{2.5})$ off-line optimal task assignment algorithm.

The proof for Theorem 8 is constructive, but requires the entire POEG to be available. The algorithm is of little use, therefore, for on-the-fly anomaly detection. However, it can be used to perform task assignment for trace-based analysis. In fact, this is the best general task assignment algorithm (since there is also a reduction from maximum bipartite matching). Any improvements in maximum bipartite matching will lead to a similar improvement in the off-line task assignment algorithm.

On-the-fly anomaly detection requires an *on-line* task assignment algorithm that makes decisions based on the current state of the POEG. Unfortunately, there is no optimal on-line task assignment algorithm. Theorem 9 gives a lower bound on the number of tasks required in the worst case.

> **Theorem 9:** A lower bound on the largest number of tasks needed for any on-line task assignment algorithm on POEGs is $\frac{3T}{2} - log(T)$.

It is important to note that whether or not a task assignment is optimal does not affect the correctness of the concurrency relationship represented by the parent vectors; suboptimality only increases the overheads associated with parent vector maintenance.

To illustrate the difficulty of performing an optimal on-line task assignment, consider the POEG in Figure 3.3. At this stage in the execution, block b_1 has been created but is not yet assigned. After blocks 1_5 and 2_4 coordinate synchronously, free tasks 4 and 6 are available for assignment to children of both 1_5 and 2_4. In particular, block b_1 can be assigned either task 4 or 6. However, depending on the future execution of the program, either choice may be suboptimal.

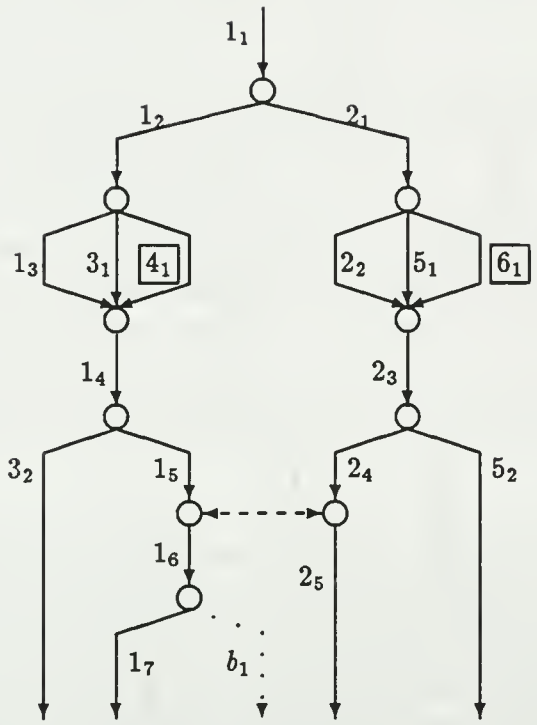


Figure 3.3: Partial Task Assignment

Suppose b_1 is assigned task 6. At some time in the future, block 5_2 could create two new threads, b_2 and b_3. This POEG is shown in Figure 3.4. Both b_2 and b_3 are concurrent with block 4_1 and therefore cannot use task 4. A new task must be generated to assign one of b_2 or b_3, the other can be assigned task 5_3. This forces seven tasks to be used in the assignment even though the graph has a maximum concurrency of six. (A symmetric scenario holds if b_1 is assigned task 4.)

The following definitions are used in the proofs of Theorem 7 and supporting Lemma 2 and Lemma 3. Let $G = (V, E)$ be a POEG, $V_1 \subset V$ and $V_2 = V - V_1$, and let O be a set of block edges from vertices in V_1 to vertices in V_2. O is an *oriented cut* if and only if all edges in E that have one end point in V_1 and one in V_2 are directed from V_1 to V_2. A *path cover* is a set of paths from the root to the leaf node which together traverse every block edge in the graph at least once.

Lemma 2 *A set of block edges are concurrent if and only if they are in an oriented cut.*

Proof.

$\Longleftarrow$ Suppose, for the sake of contradiction, that there are two edges (v_i, a_0) and (a_k, v_j) in an oriented cut and (v_i, a_0) is an ancestor of (a_k, v_j). From the definition of an oriented

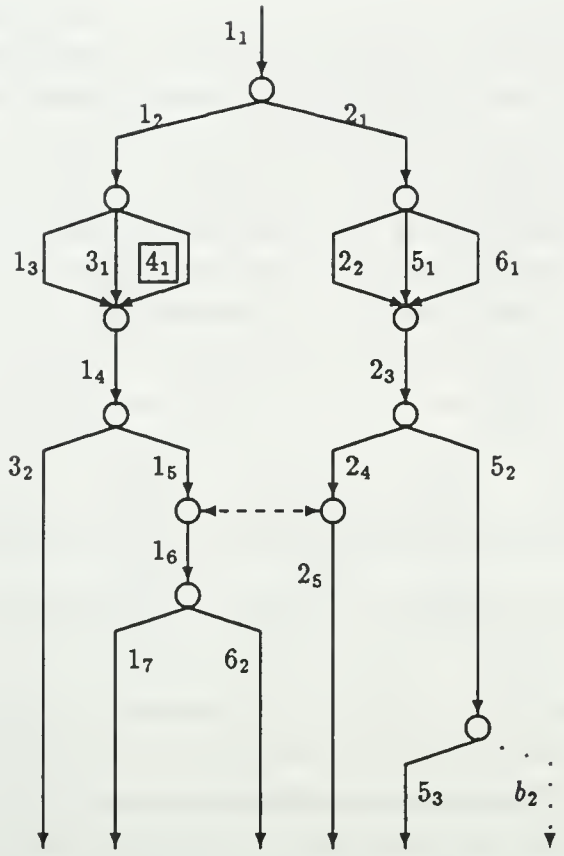


Figure 3.4: Partial Task Assignment

cut we know that v_i and $a_k \in V_1$ and a_0 and $v_j \in V_2$. Consider all of the vertices $a_1 \ldots a_k$ on a path from a_0 to v_j. If $a_1 \in V_1$, the edge (a_0, a_1) would go from V_2 to V_1; therefore, a_1 must be in V_2. By continuing this argument we can deduce that a_{k-1} must be in V_2. However, from our original assumption we know that a_k is in V_1. Therefore, if two non-concurrent blocks are in the cut there must be an edge from V_2 to V_1.

$\Longrightarrow$ Suppose, for the sake of contradiction, that there is a set of concurrent blocks B that is not a subset of any oriented cut. There must be a maximal subset B' of B such that all blocks in B' are in at least one oriented cut O, but B' is never in a cut with some block $b \in B - B'$. Since O is a cut, the set of blocks $A = O - B'$ must contain only ancestors or only descendants of b. We know there are no paths between A and any other block in O (since all blocks in a cut are concurrent) or between any block in B' and b. If A contains ancestors of b, we can move A and all blocks on paths from A to b to V_1 and make b part of O; if A contains descendants of b, we move A and all blocks on paths from b to A to V_2 and make b part of O. Because we can add b to O, a contradiction has been reached. □

Lemma 3 *The number of paths needed to cover a POEG is equal to its maximum concurrency.*

Proof. (By induction on the maximum concurrency T of a POEG G) :
Base Case ($T = 1$): All blocks in G have a single child, so G can be covered with one path.
Induction Hypothesis: Suppose that the lemma holds for all POEGs with $T < i$; prove that all POEGs with $T = i$ can be covered with i paths. Create the leftmost path L from the root to the leaf block and let M be the set of blocks in L that appear in cuts of size i. We define a *null* vertex to be any vertex with one parent block and one child block. We will modify G to obtain a new POEG G' with maximum concurrency $i-1$ by deleting all blocks in M.

In the first step we modify every vertex that is the head or tail of a block $b = (v_p, v_c) \in M$. There can never be more than one path through b; this follows directly from Lemma 2 and the fact that a path cannot traverse concurrent blocks. Therefore, if v_p is a receiver vertex, its associated coordination edge will never be traversed and can be deleted thereby changing v_p into a null vertex. We know that v_p cannot be a join vertex: otherwise, the parent blocks of b would be part of a subgraph with maximum concurrency less than i. By symmetric arguments, if v_c is a sender vertex then we delete the coordination edges and change it to a null vertex, and we know that it cannot be a fork vertex. After making this transformation, every block edge in M has a parent vertex that is either a sender, null, or fork vertex and a child vertex that is either a receiver, null, or join vertex.

The following properties hold for every maximal sequence S of blocks in M:

- S begins with either a fork or receiver vertex: if S begins with a null vertex v, the parent block of v would be in a directed cut of size at least i and be part of S.
- Likewise, S ends with either a join or sender vertex.
- Every internal vertex $v \in S$ is a null vertex: v cannot be either a sender or fork vertex because it is the child vertex of some block in M, and v cannot be either a receiver or join vertex because it is the parent vertex of some block in M.

Each sequence S in M is deleted from G creating a new POEG G' with maximum concurrency $i-1$. Because each S starts with a vertex with multiple child blocks, ends with a vertex with multiple parent blocks, and has no internal coordination edges, G' remains well formed. By our induction hypothesis, G' can be covered with $i-1$ paths and therefore the lemma holds for all i. □

Theorem 7 *The minimum number of tasks needed to perform a valid assignment to a POEG is equal to the maximum concurrency of the POEG.*

Proof. Let A be the number of tasks needed to perform the smallest valid assignment of a POEG and T the maximum concurrency of the POEG. From Lemma 3 we know that the number of paths C needed to cover a POEG is equal to T. All blocks on a path can be assigned to the same task; by definition they are related by an ancestor/descendant relationship and by Theorem 1 are not concurrent. Thus, the number of tasks needed for a valid assignment of the POEG is less than or equal to the size of the path cover, so that $A \leq C = T$. However, each concurrent block needs a unique task so that $A \geq T$. Since $A \leq T$ and $A \geq T$, it must be the case that the size of the smallest valid task assignment is equal to the maximum concurrency of the POEG. □

Theorem 8 *The complexity of performing an off-line optimal task assignment is equivalent to the complexity of maximum bipartite matching.*

Proof. The proof of Theorem 8 consists of a linear time reduction of optimal task assignment to maximum bipartite matching and a linear time reduction of maximum bipartite matching to optimal task assignment.
$\Longleftarrow$ We construct a bipartite graph $G_B = (\{P, C\}, E)$ for a given POEG G such that:

$|P| = |C|$ = the number of blocks in the POEG, and
$(p_i, c_j) \in E$ if and only if $p_i \in P$, $c_j \in C$ and b_i is an ancestor of b_j in G.

There is an ancestor set associated with each vertex v in the POEG that is computed by performing a topological traversal of the POEG. If a block b ends in a fork or join vertex v, $O(B)$ work is required to update the information associated with v. If b ends in a sender vertex, it performs a union of its ancestor information with the ancestor information of the other senders of that coordination point. This composite ancestor set is used by each of the receiver blocks of the coordination point. For programs that meet the coordination constraint[4], the total amount of work required to compute the transitive closure is $O(B^2)$.

A matching M consists of $|M|$ edges where:

- M_C denotes the end points in M from vertex set C,
- M_P denotes the end points in M from vertex set P,
- The two endpoints of an edge in M are *partners*,
- The two vertices that correspond to the same block (e.g. p_i and c_i) are *mates*.

Two properties follow from these definitions:

[4]For POEGs that do not meet the coordination constraint, the amount of work required is bounded by $O(B^2 \times T)$. If one wants to consider the problem of task assignment to general partial orders, the amount of work required to compute the ancestor sets (e.g. the transitive closure of the partial order) is equivalent to performing matrix multiplication.

1. Every matching M of G_B corresponds to a valid task assignment which uses $|C|$ - $|M|$ tasks. Each vertex $\notin M_C$ is assigned to a unique task; each vertex $\in M_C$ is assigned to the task associated with the mate of its partner vertex with one higher version number. This assignment algorithm is valid because each task is assigned to a set of non-concurrent blocks, and every block has a task assigned to it. Moreover, the number of tasks required for the assignment is equal to $|C| - |M|$.

   Figure 3.5 shows an example matching. This would result in in b_1 being assigned 1_1

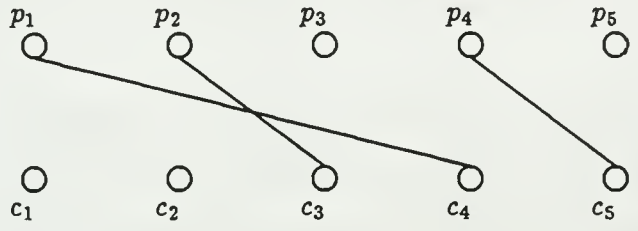


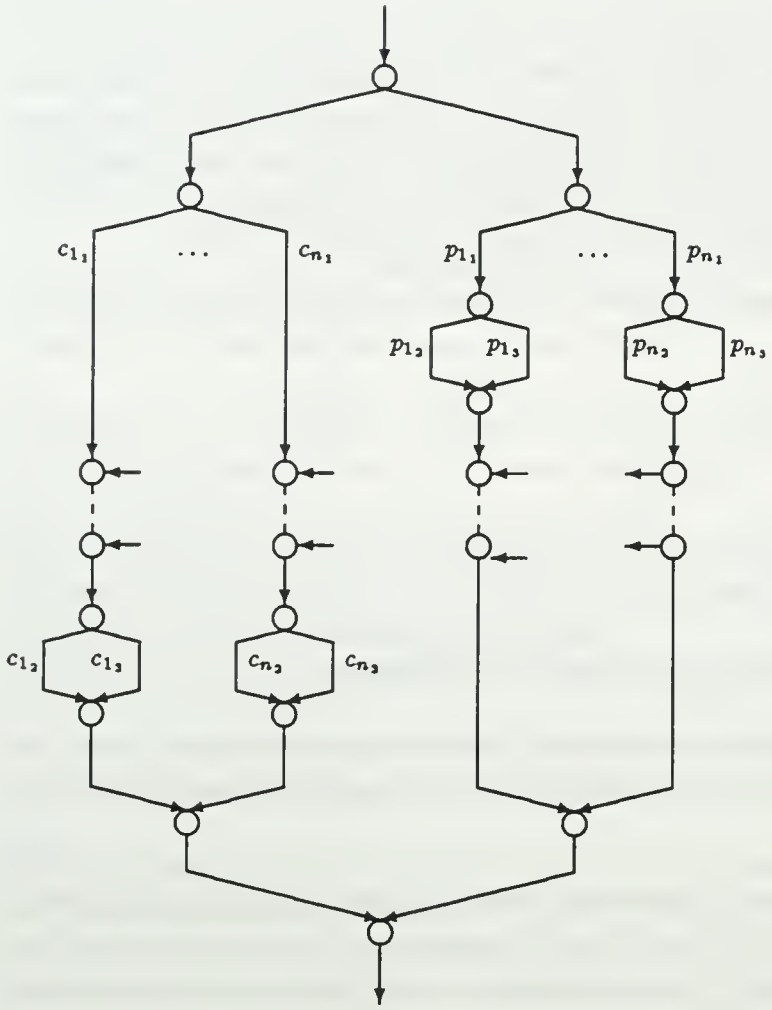
Figure 3.5: Example Bipartite Matching

   and b_2 assigned 2_1 since they are not in M_C. Block b_3 would be assigned 2_2 since the mate (c_2) of b_3's partner (p_2) was assigned task 2_1. Similarly, b_4 would be assigned 1_2 and b_5 assigned 1_3.

2. Every valid task assignment which uses T tasks has a corresponding matching M of G_B of size $|C| - |T|$. For each task t add to M every edge in G_B which connects p_{t_i} and $c_{t_{i+1}}$ (t_j denotes the j^{th} block assigned to task t). This is a matching because each block is assigned only one task and there is an edge between two nodes whenever it is required. The only vertices in C which are not included are the first blocks assigned to each task (e.g. t_0 for each task t) and hence $|M|$ is $|C| - |T|$.

It follows directly from 1 and 2 that by creating a maximum matching we obtain an optimal task assignment.

$\Longrightarrow$ We are given an instance of a maximum matching problem for bipartite graphs with graph $G_B = (\{P, C\}, E)$ such that $|P| = |C| = n$, and for all edges $(p, c) \in E$, $p \in P$ and $c \in C$. We can transform this into a problem of task assignment by creating a parallel program with the corresponding POEG G as is shown in Figure 3.6. G initially creates two sets of parallel threads: $p_1 \ldots p_n$ and $c_1 \ldots c_n$. Each p_{i_1} creates two parallel blocks p_{i_2} and p_{i_3} that are sequentially followed by a sequence of blocks that are separated by sender vertices. Each c_{i_1} is followed by a sequence of blocks that are separated by receiver vertices and then two parallel blocks c_{i_2} and c_{i_3}. There are asynchronous coordination edges from thread p_i to thread c_j if and only if there is an edge from vertex p_i to vertex




```
parallel case
    doall c := 1 to n
        for all p do
            if (p, c) ∈ E then
                receive(p)
            endif
        end for
        doall j := 1 to 2
            null statement
        endall
    endall
parallel
    doall p := 1 to n
        doall j := 1 to 2
            null statement
        endall
        for all c do
            if (p, c) ∈ E then
                send(c)
            endif
        end for
    endall
parallel end
```

Figure 3.6: Example POEG

c_j in G_B. The number of block edges in G is $5C + 5P + 2E + 6$ and the number of coordination edges is E so that this transformation requires $O(C + P + E)$ space and work.

The number of tasks required to assign G if the coordination edges are ignored is $4n$. Consider some task assignment A. Assume for all i that c_{i_1} and c_{i_2} are assigned the same task and that p_{i_1} and p_{i_2} are assigned the same task. (Which of c_{i_2} or c_{i_3} is assigned the same task as c_{i_1} does not affect the optimality of an assignment.) If $|A| < 4n$, then a task T was assigned to both p_{i_3} and c_{j_3} for some i and j.

Consider any maximum matching M for G_B. There is a corresponding optimal task assignment A that uses $4n - |M|$ tasks. In A, block c_{j_3} is assigned to the same task as

block p_{i_3} if and only if there is an edge (p_i, c_j) in M. If there was a better task assignment in G it would result in a large matching in G_B. Conversely, an optimal assignment A corresponds to a matching M of size $4n - |A|$ in which an edge (p_i, c_j) is included in M if and only if block c_{j_3} is assigned to the same task as block p_{i_3} in A. Therefore, the amount of work required to perform an optimal task assignment is an upper bound on the work required to create a maximum matching. □

Corollary 2 *There exists an* $O(B^{2.5})$ *off-line optimal task assignment algorithm.*

Proof. This follows directly from Theorem 8 and results by Hopcroft and Karp [HK73]. □

Lemma 4 *There exists a POEG with maximum concurrency* $2^{i+1} - 1$ *such that at least* $2^{i+1} + 2^{i-1} - 2$ *tasks are required by any on-line valid task assignment.*

Proof. (By adversary argument): Consider the POEG in Figure 3.7. It has three

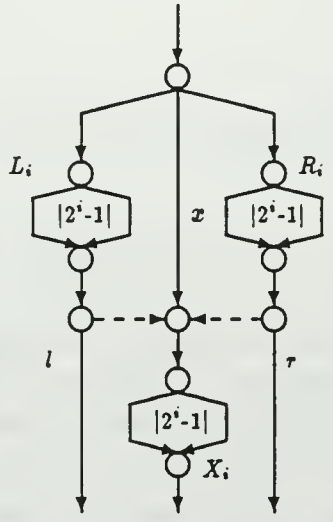


Figure 3.7: POEG at time t

branches: two have a fork-join set containing $2^i - 1$ parallel blocks and one is a single thread. The set X_i is a child of both L_i and R_i. The maximum concurrency of this graph is therefore $2^i - 1 + 2^i - 1 + 1 = 2^{i+1} - 1$.

A task is *reachable* from a block b if and only if it was last assigned to an ancestor of b. Let $avail_b$ be the number of free tasks reachable by block b. We will order the execution such that all of the blocks in X_i are assigned before either block l or r. The tasks last assigned to block x and the blocks of L_i and R_i are reachable from the blocks

in X_i. After the task assignment of the blocks of X_i, the number of tasks reachable from l and r is equal to the sum of the sizes of L_i and R_i less the tasks used from them during the assignment; namely, $avail_r + avail_l = 2 \times (2^i - 1) - (2^i - 2) = 2^i$. Therefore, either $avail_l$ or $avail_r$ must be less than or equal to 2^{i-1}.

If $avail_l \leq 2^{i-1}$, we create a subgraph of l that is similar to the original graph except that each fork-join set contains $2^{i-1} - 1$ parallel blocks (shown in Figure 3.8a); a sym-

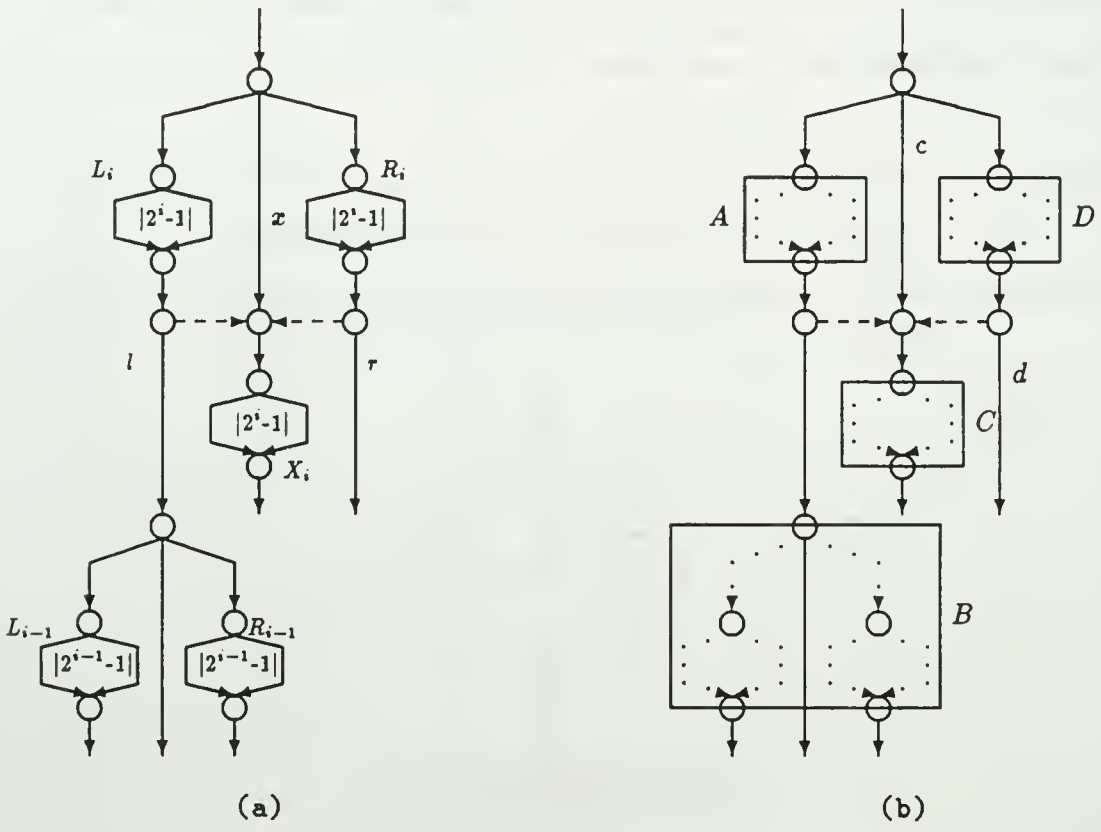


Figure 3.8: POEG at time $t + 1$ if $avail_l \leq 2^{i-1}$

metric construction is used if $avail_r \leq 2^{i-1}$. We first show that this modification does not change the maximum concurrency of the POEG. There are four groups of blocks that form maximally oriented cuts for each of the three branches of the graph (as is seen in Figure 3.8b); each group has maximum concurrency of $2^i - 1$. The three oriented cuts of the entire graph that pass through more than one of these groups–$\{A, D, c\}$, $\{B, D, c\}$, and $\{B, C, d\}$–all have maximum concurrency $2^{i+1} - 1$ and therefore, the maximum concurrency of the graph remains $2^{i+1} - 1$.

The number of tasks needed for the task assignment of the subgraph is equal to its maximum concurrency. Hence, the number of *extra* tasks required is the maximum concurrency less the number of tasks reachable from l; namely, it is at least $(2^i-1)-(2^{i-1})$. Therefore, the total number of tasks required for a valid assignment to the entire graph is at least $(2^{i+1}-1)+(2^{i-1}-1)$. □

Theorem 9 *A lower bound on the largest number of tasks needed for any on-line task assignment algorithm on POEGs is* $\frac{3T}{2} - log(T)$.

Proof. We start with the graph in Figure 3.7 as described in the proof of Lemma 4. The blocks in the subgraph have no reachable tasks, and hence Lemma 4 can be applied recursively to this subgraph. If we start with an initial fork-join set size of $2^i - 1$, we can create $i-1$ nested subgraphs; the fork-join sets of the last subgraph consist of a single parallel block. This construction never increases the maximum concurrency T of the graph, which is $2^{i+1}-1$, and the number of extra tasks required is at least:

$$\sum_{j=1}^{i-1} 2^{i-j} - 1 = \sum_{j=1}^{i-1} 2^j - (i-1) = 2^i - (i+1) \geq \frac{T}{2} - log(T)$$

Therefore, the total number of tasks needed to validly assign this POEG is at least $\frac{3T}{2} - log(T)$. □

3.4 On-line Task Assignment Algorithms

This section presents a basic algorithm for performing task assignment and several other variations. The algorithms attempt to limit the number of tasks in an assignment by using a "most recently used" (MRU) heuristic. The intuition behind this heuristic is that if a block a can validly be assigned to the free task that was last assigned to block bR, then a can also validly be assigned to any free task that was last assigned to an ancestor of b. A *free task dag* stores information about the tasks that are available for reassignment as well as the concurrency relationship of the blocks that were last assigned those tasks.

3.4.1 MRU Task Assignment Algorithm

Suppose a set of tasks T are currently not in use and block b needs to be assigned a task. The MRU algorithm will assign b a task $t_j \in T$ such that the following two conditions are met:

1. Task t_j was last assigned to a block a that is a direct ancestor of b, and
2. No task in T was last assigned to a descendant of a.

The first condition guarantees that the assignment is valid (e.g. no task is assigned to two concurrent blocks). The second condition enforces a "most recently used" heuristic for limiting the number of tasks used in the assignment.

To illustrate the MRU algorithm consider the POEG in Figure 3.9a. Blocks $b_1 \ldots b_5$

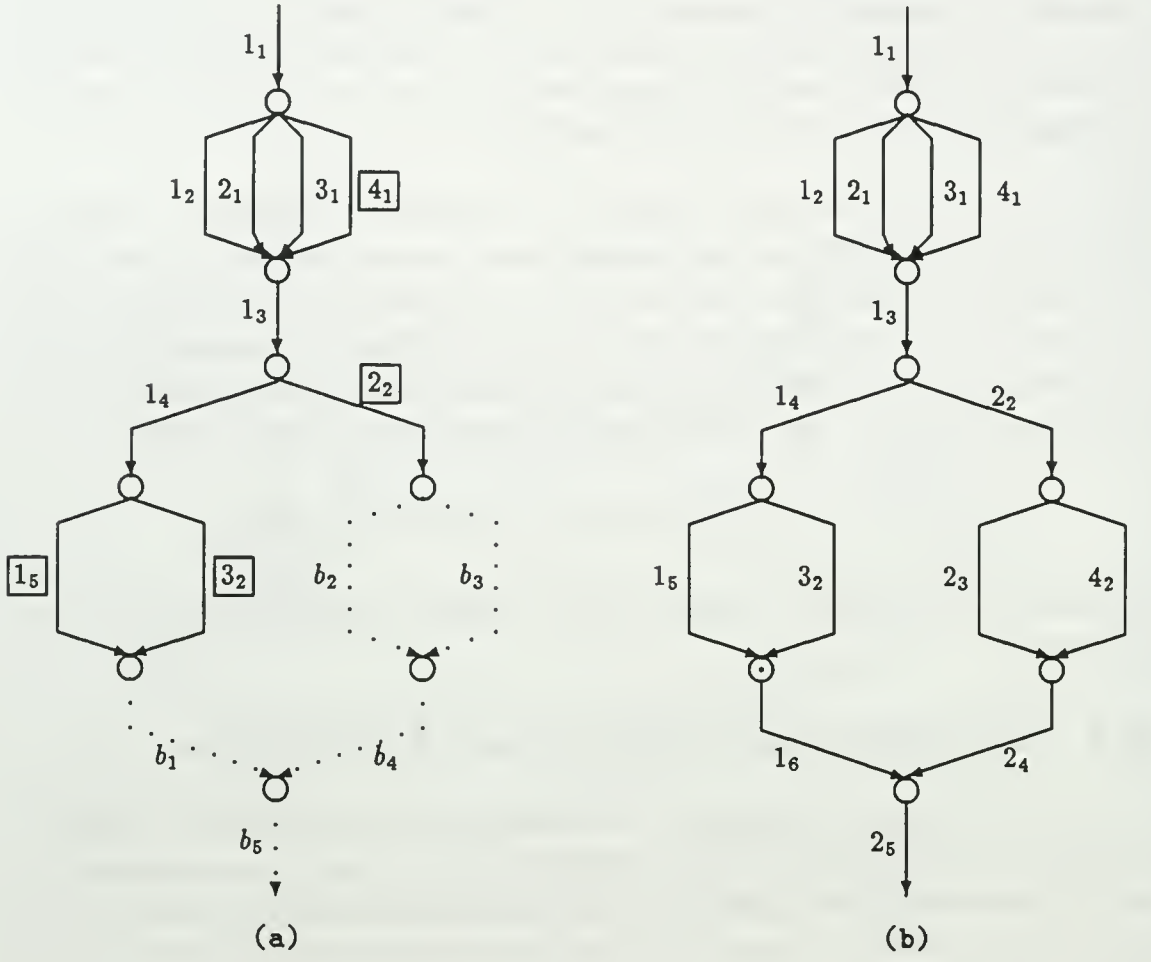


Figure 3.9: Optimal Task Assignment to a POEG

are unassigned and tasks 1, 2, 3, and 4 are available for reassignment. Suppose b_1 is the next block to be assigned. Block b_1 may be validly assigned either task 1, 3, or 4; block b_1 cannot be assigned to task 2 since b_1 is concurrent with block 2_2. Based on the MRU heuristic, block b_1 will be assigned either task 1 or 3. Figure 3.9b shows an optimal assignment made by the MRU task assignment algorithm in which block b_1 is assigned task 1. A symmetric assignment results if block b_1 is assigned task 3. However, if block b_1 is assigned task 4 the overall assignment cannot be optimal: a new task will have to

be created to assign to either b_2 or b_3.

The MRU algorithm can be implemented at run-time by maintaining a dag of free tasks that are available for reassignment. The internal nodes of the free task dag are associated with fork and join operations, and the leaf nodes are associated with currently executing blocks. A set of free tasks is associated with each node. The *outnode* set of a node n in a free task dag is the set of nodes to which n points. The *entry point* of an executing block b is the node in the free task dag that is associated with b. Since we assign tasks in the reverse order of the execution flow, the edges in the free task dag point towards the root of the dag.

When a block b is created after operation O, an entry point n_b is created for b, an edge is added from n_b to the node n_O associated with operation O, and a task is assigned to b. To assign a task to a block b, the free task dag is traversed starting at n_b until a node with a non-empty free task set is found. When a block b terminates at a fork or join operation O, it updates the node n_O associated with O by adding its free task set and outnodes to node n_O and deleting entry point n_b. It also adds its task to the free task set of node n_O.

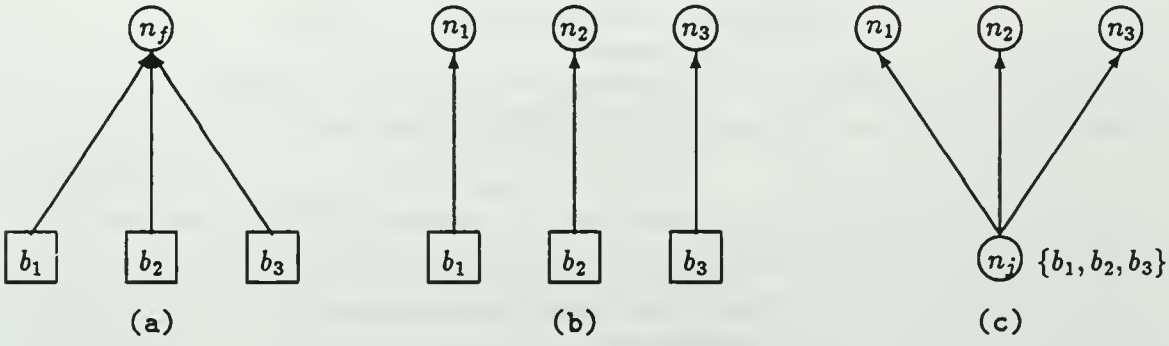


Figure 3.10: Construction of Free Task Dag

Figure 3.10 shows the structure of the free task dag after various operations. (An entry point node is represented by a box and an operation node is represented by a circle.) The state of the free task dag after blocks b_1, b_2, and b_3 are created by fork operation f is shown in Figure 3.10. Figures 3.10b and 3.10c respectively show the free task dag just before b_1, b_2 and b_3 terminate, and immediately after they terminate in join operation j.

Unfortunately, when the free task dag is maintained in this manner, it will grow in proportion to the length of the execution. In addition, task assignment is computationally expensive if long paths of nodes with empty free task sets are traversed before an available free task is found. The following path compression operations–*subsume* and

collapse–keep the free task dag compact and task assignment efficient by deleting unneeded nodes. Lemma 6 proves that by performing these two optimizations whenever possible, the structure of the free task dag is restricted to being a tree. The means that every node has a single outnode. In addition, every internal node has a non-empty free task set. These two conditions simplify the code for the management of the free task dag. The algorithms for maintaining the free task dag when a block terminates at fork or join operations and for assigning tasks to newly created block are shown in Algorithm 3.7. When a block performs a coordination operation, it simply increments its version number.

```
procedure Fork(b, O, c_1 ... c_W)
    n_O := n_b
    n_O.free-tasks := n_O.free-tasks ∪ task
    for all children blocks c_1 ... c_W do
        make and initialize node n_{c_i}
        n_{c_i}.outnode := n_O
        n_O.innode := n_O.innode + n_{c_i}
    endfor
end procedure

procedure Join(b, O, c_j)
    c = n_b.outnode
    if n_O = null node then
        make and initialize node n_O
        n_O.outnode := c
        c.innode := c.innode + n_O
        n_{c_j} := n_O
    endif
    c.innode := c.innode − n_b
    n_O.free-tasks := n_O.free-tasks ∪ task
    remove node n_b
end procedure

procedure Assign-Task(n_b, task)
    if n.free-tasks ≠ ∅ then
        pick task from n.free-tasks
    else if n.outnode.free-tasks ≠ ∅ then
        pick task from n.free-tasks
    else task := new task
end procedure
```

Algorithm 3.7: Task Assignment and Free Task Management

Whenever a free task set of an internal node n becomes empty, the dag is *collapsed* by deleting n from the dag. (The only exception is the root node, which is never collapsed.)

All nodes that point to n are modified to instead point to the outnode of n. This guarantees that every internal node in the free task dag has at least one free task associated with it. A collapse operation can be performed after a task is assigned. Whenever an internal node n is the outnode of a single node p, n is *subsumed* by p by adding the free task set of n to the free task set of p, deleting n and adjusting the edges as appropriate. A subsume operation can be performed whenever a node is deleted from the free task dag. Figure

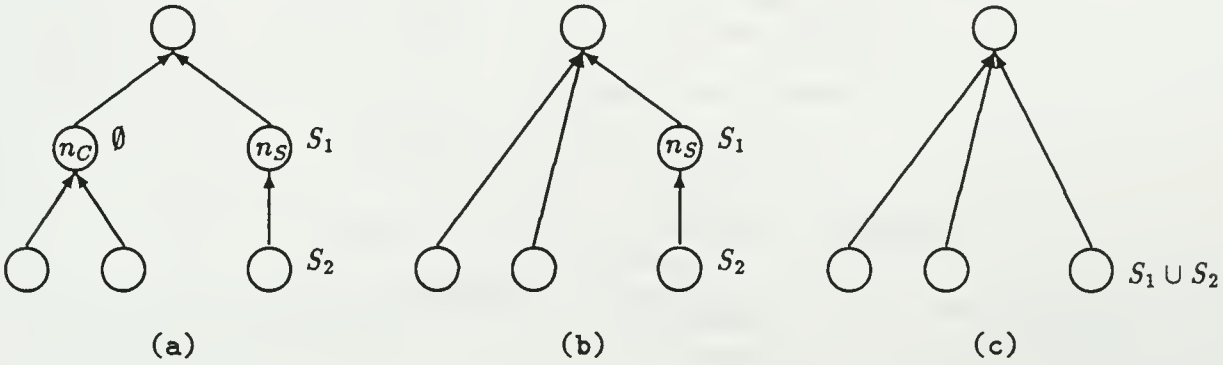


Figure 3.11: Path Compression Optimizations

3.11 shows the changes to the free task dag after node n_C is collapsed and then node n_S is subsumed. The algorithms for subsuming and collapsing the free task dag are shown in Algorithm 3.8.

To illustrate the MRU algorithm, consider Figure 3.12. Figure 3.12 shows the state of the free task dag at specific stages in the execution of the example program in Figure 3.9. Before the first fork operation there are three free tasks; namely, 2, 3 and 4. The corresponding state in the free task dag is shown in Figure 3.12a. Figure 3.12b shows the free task dag when the program is in the state in Figure 3.9a. At this point in the execution, block b_2 can be assigned blocks 2 or 4. (Block b_1 can be validly assigned to task 1, 3 or 4.) Because task 2 was more recently used than task 4, block b_2 is assigned task identifier 2_3. Similarly, block b_1 is assigned task identifier 1_6. (Block b_1 could equivalently be assigned task identifier 3_3). Figure 3.12c shows the free task dag when the program is in the state in Figure 3.9b. The free task dag has been collapsed and subsumed leaving a single node.

3.4.2 Complexity of the MRU Algorithm

The theorems in this section prove the complexity of the MRU algorithm.

> **Theorem 10:** The number of tasks in a task assignment performed by the MRU algorithm is T'.

```
procedure Check-Subsume(n)
    if |n.innode| = 1 then
        c := n.outnode
        p := n.innode
        p.free-tasks := p.free-tasks ∪ n.free-tasks
        p.outnode := c
        c.innode := c.innode + p − n
        remove node n
    endif
end procedure

procedure Check-Collapse(n)
    if n.free-tasks = ∅ and n ≠ root then
        c := n.outnode
        c.innode := c.innode − n
        for all p ∈ n.innode do
            p.outnode := c
            c.innode := c.innode + p
        end for
        remove node n
    end if
end procedure
```

Algorithm 3.8: Free Task Path Compression

Theorem 11: The size of the free task dag used by the MRU algorithm is $O(T + T')$.

(Recall that T denotes the maximum concurrency of the POEG and T' denotes the maximum concurrency of the POEG with coordination edges deleted.) Theorem 10 proves that T' tasks are used by the MRU algorithm to perform a valid assignment. Therefore, the MRU algorithm is guaranteed to be optimal for programs that do not include any coordination or unconstrained fork-join operations. Theorem 11 proves that the size of the free task dag is bounded by $O(T + T')$. As with parent vector management, this cost is a worst case measurement and the average case cost is much lower.

Theorem 12 proves that the average cost per block is proportional the maximum level of nesting of fork-join constructs, N.

Theorem 12: The average cost per block of performing the MRU algorithm is $O(N)$.

A direct consequence of Theorem 12 is that $O(1)$ work is performed for task assignment for non-nested parallel constructs. Regardless of the level of nesting, if the number of threads created in a parallel construct is always greater than twice the level of nesting,

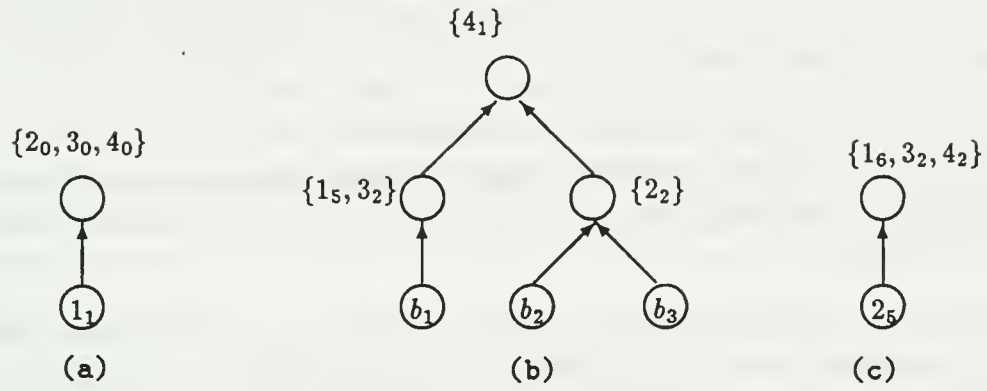


Figure 3.12: Stages in Free Task Dag for POEG in Figure 3.9

(e.g. $W > 2N$), the number of nodes in the free task dag is $O(\frac{B}{N})$ and the average work for a task assignment is $O(1)$.

Lemma 5 *The height of a node in the free task dag can never increase.*

Proof. Only the subsume and collapse operations change the height of a node in the free task dag. In both cases, the nodes that point to the deleted node n are modified to point to the outnodes of n rather than to n. Because the dag is acyclic, this cannot increase the height of a node in the dag. □

Lemma 6 *The free task dag for the MRU algorithm is always a tree in which every entry point n_b is at height at most $N + 1$ where N is the fork-join nesting level of block b.*

Proof. Proof by induction on the level of nesting of parallel constructs:
Base Case (N=1): Before the first fork operation there is a single node n_r in the free task dag that is the entry point of the current block. The fork operation adds the task of the current block to n_r and creates k entry point nodes $n_1 \ldots n_k$–one for each block created by the fork operation–that have one outnode n_r (and are thus at height 2). At the join operation, the entry point n_j of the block after the join operation is created, and the outnodes of $n_1 \ldots n_j$ become the outnodes of n_j. Node n_r is the sole outnode of $n_1 \ldots n_k$; therefore n_j has one outnode, n_r. After $n_1 \ldots n_j$ are deleted, the only node that points to n_r is n_j, and hence n_j will subsume n_r resulting in a tree of height 1. Any subsequent fork-join operations perform the same transformations to the graph.
Induction Hypothesis: Assume that the tree structure holds for all POEGs with fork-join nesting at most i: prove that it holds for all POEGs with nesting $i + 1$. The entry point n_f of the block performing a fork operation of nesting $i + 1$ is at height at most $i + 1$. We

then create k new entry points $n_1 \ldots n_k$ that have a sole outnode, n_f (and therefore are at height $i+2$). There are two cases that can occur during task assignment of the blocks $b_1 \ldots b_k$ created by the fork operation:

1. If there are enough tasks associated with n_f to assign $b_1 \ldots b_k$, then n_f remains the outnode of $n_1 \ldots n_k$. After we perform the join, the entry point n_j is the sole node that points to n_f and will subsume it. Therefore, the height of n_j is equal to the height of n_f which is at most $i+1$.

2. If there are not enough tasks associated with n_f to assign $b_1 \ldots b_k$ then after some task b_j is assigned, node n_f will be collapsed, and all of the nodes that point to it (namely, $n_1 \ldots n_k$) will be modified to point to the outnode of n_f. By the induction hypothesis, n_f has exactly one outnode that is at height at most i. Path compression can occur many times during the task assignment process, but in all cases the height of nodes $n_1 \ldots n_k$ decrease, and they all have the same outnode. At the join, the outnode of $n_1 \ldots n_k$ is made the outnode n_j, and therefore n_j is at height at most $i+1$.

In both cases the dag is always a tree in which each entry point is at height at most one more that its level of nesting. □

Theorem 10 *The number of tasks in a task assignment performed by the MRU algorithm is T'.*

Proof. For a given POEG G there is a simplified POEG, G', which is G with all coordination edges removed. The maximum concurrency of G' is equal to the maximum concurrency of G with all coordination edges removed, namely $T_{G'} = T'_G$.

The optimality of the assignment performed by the MRU algorithm for G' is based on three properties:

1. Every block that can select a task from node n can also select a task from the outnode of n, but not vise versa.

2. The selection among the tasks in a given free task set does not affect the optimality of the assignment.

3. The free task dag correctly represents all ancestor tasks for each block b in G' (no block in G' has any indirect ancestors).

Only two choices are made in performing a task assignment: which free task set F to use, and which task to use from F. By Lemma 6, every node has a single outnode, and there is always at most one leaf associated with each block b. Because of property 1, it is always

optimal to assign all of the tasks from a leaf node n in the free task dag before assigning a task from the outnode of n. Therefore, the first choice optimally selects the appropriate free task set F. Because of property 2, the choice of task within the free task set F does not affect the optimality of an assignment. Because of property 3, the task assignment performed must be optimal and by Theorem 7 uses $T_{G'} = T'_G$ tasks. Since the assignment performed for G is identical to the assignment performed by G', the assignment to G must also use T'_G tasks. □

Theorem 11 *The size of the free task dag used by the MRU algorithm is* $O(T + T')$.

Proof. There can be at most T leaf nodes since every leaf node is the entry point of an executing block, and T is the maximum concurrency of the program. Therefore, the number of nodes and edges in the free task dag is $O(T)$. The size of the free task sets is $O(T')$ since, by Theorem 10, there are at most T' tasks and therefore, at most T' unassigned tasks. Hence, the total size of the free task dag is bounded by $O(T + T')$. □

Theorem 12 *The average cost per block of performing the MRU algorithm is* $O(N)$.

Proof. Consider the operations that may have to be performed for each block. An assignment and (possibly) a collapse operation are performed when a block is created; a fork or join and (possibly) a subsume operation is performed when a block terminates. The assignment, join, and subsume operations each performs $O(1)$ work. Although the fork operation performs $O(W)$ work (where W is the number of threads created by the fork), this cost is an average of $O(1)$ work for each of the W new threads. The collapse operation performs work linear in the number of nodes that point to the collapsed node. However, a collapse operation is not performed by every block.

The work performed per block for collapse operations can be calculated by computing the work performed over the lifetime of each node to update its outnode information when other nodes are collapsed. This figure is summed over all nodes in the free task dag, F, and then divided by the number of blocks, B.

The only nodes whose collapse can affect a node n are the nodes on the path from n to the root of the dag. Whenever the outnode of n is collapsed, $O(1)$ work is done updating n, and the height of n is decreased by one. By Lemma 6, a node n at level of nesting l is originally at height at most $l + 1$. Thus, the total amount of work done updating outnode information for the entire dag is:

$$O(\sum_{i=1}^{F} height(n_i)) \leq O(FN)$$

(where N is the maximum level of nesting). Since $O(F) = O(B)$, the average amount of work performed per block is $O(N)$. □

3.4.3 Modified MRU Task Assignment Algorithm

The MRU assignment algorithm uses T' tasks to perform a valid task assignment. However, the size of the optimal assignment is equal to the maximum concurrency of the graph T. Since T can be less than T', the MRU algorithm is only guaranteed to be optimal for programs that contain no coordination[5]. There is the potential for reducing the size of the task assignment of programs with coordination.

For example, Figure 3.13 shows an optimal task assignment for a POEG with a coordination edge. The number of tasks used by the optimal assignment is equal to the maximum concurrency of four. Because the MRU algorithm disregards the ancestors obtained from

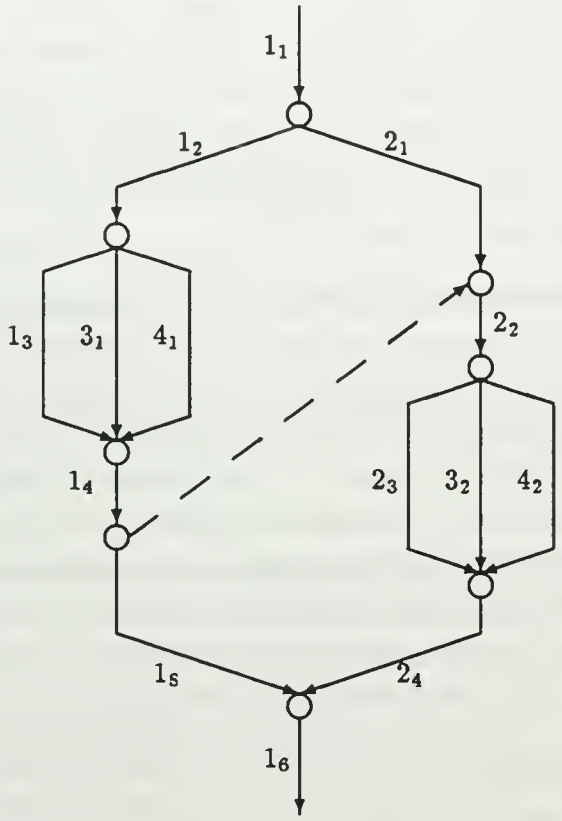


Figure 3.13: POEG with Coordination

the coordination operations, it must use six tasks to perform a valid assignment.

The complexity of performing an optimal task assignment stems from the presence of coordination. After a block b coordinates, any of the tasks that were last assigned to either direct or indirect ancestors of b can be validly assigned to b. In order to use tasks

[5]The simple MRU algorithm is also suboptimal for programs whose thread creation and termination operations are not parallel constructs.

assigned to indirect ancestors, the algorithms for managing the free task dag must be modified to include information about coordination operations.

To do so, the structure of a node in the free task dag is extended. Each node has three types of outnodes: a tree outnode, direct outnodes and indirect outnodes. The *direct outnodes* are outnodes from fork and join operations in parallel constructs; the *indirect outnodes* are outnodes obtained from coordination operations and unconstrained fork and join operations. (If a node is in both the direct and indirect outnode of another node, it is deleted from the indirect set.) The *tree outnode* is one of the direct outnodes. Direct and indirect outnodes are always propagated to the leaf nodes so that internal nodes have only a single tree outnode.

To represent coordination, a node is associated with each coordination point. This node accumulates information about all sender blocks and is then used to modify the entry point nodes of all receiver blocks. In particular, the entry point of a receiver block is modified to have as indirect outnodes the tree, indirect and direct outnodes of the entry points associated with its indirect parents. Algorithm 3.9 shows the algorithms for maintaining the free task dag when a coordination operation is performed.

```
procedure Coordinate-Sender(b, O)
    make and initialize node new
    new.tree := n_b
    new.direct := n_b.direct
    new.indirect := n_b.indirect
    n_O.indirect := n_O.indirect ∪ n_b.indirect ∪ n_b.direct ∪ n_b
    for all n ∈ n_b.direct ∪ n_b.indirect do
        n.innode := n.innode + n_O + new − n_b
    endfor
    n_b.innode := n_O + new
    n_b := new
end procedure

procedure Coordinate-Receiver(b, O)
    n_b.indirect := n_b.indirect ∪ n_O.indirect
    for all n ∈ n_O.indirect do
        n.innode := n.innode + n_b
    endfor
end procedure
```

Algorithm 3.9: Coordination Operation

In addition, indirect and direct outnodes are propagated to entry point nodes at fork and join operations. Thus, when a block b performs a fork or join operation O, three steps are performed: node n_b becomes the tree outnode of n_O, the indirect and direct outnodes

of n_b become the indirect and direct outnodes of n_O, and each child c of n_O transfers the indirect and direct outnodes of n_O to n_c. Algorithm 3.10 shows the modified algorithms for maintaining the free task dag.

```
procedure Fork(b, O, c_1 ... c_W)
    n_O := n_b
    n_O.free-tasks := n_O.free-tasks ∪ task
    for all children blocks c_1 ... c_W do
        make and initialize node n_{c_i}
        n_{c_i}.tree := n_O
        n_O.innode := n_O.innode + n_{c_i}
        n_{c_i}.direct := n_O.direct
        n_{c_i}.indirect := n_O.indirect
        for all n ∈ n_O.direct ∪ n_O.indirect do
            n.innode := n.innode − n_b + n_{c_i}
        endfor
    endfor
end procedure

procedure Join(b, O, c)
    if n_O = null node then
        make and initialize node n_O
        n_O.tree := n_b.tree
        n_O.direct := n_O.direct ∪ n_b.direct
        n_c := n_O
    else n_O.direct := n_O.direct ∪ n_b.direct + n_b.tree
    n_O.indirect := n_O.indirect ∪ n_b.indirect
    for all n ∈ n_b.direct ∪ n_b.indirect ∪ n_b.tree do
        n.innode := n.innode + n_O − n_b
    endfor
    n_O.free-tasks := n_O.free-tasks ∪ task
    remove node n_b
end procedure
```

Algorithm 3.10: Modified Free Task Maintenance

To illustrate free dag management in the modified MRU algorithm, Figure 3.14 shows two stages in the task assignment to the POEG in Figure 3.13. (Indirect outnodes are denoted by dashed lines.) Figure 3.14a shows the free task dag immediately before the coordination operation. At this point in the execution, tasks 3 and 4 are available for reassignment to block 1_4 and none are available to block 2_1. Figure 3.3b shows the changes made to the free task dag after the coordination between blocks 1_4 and 2_1. Free tasks 3 and 4 are now available to both blocks 1_5 and 2_2.

When indirect ancestors are included, the free task dag correctly represents all of the free tasks available for assignment to currently executing blocks. However, the free task

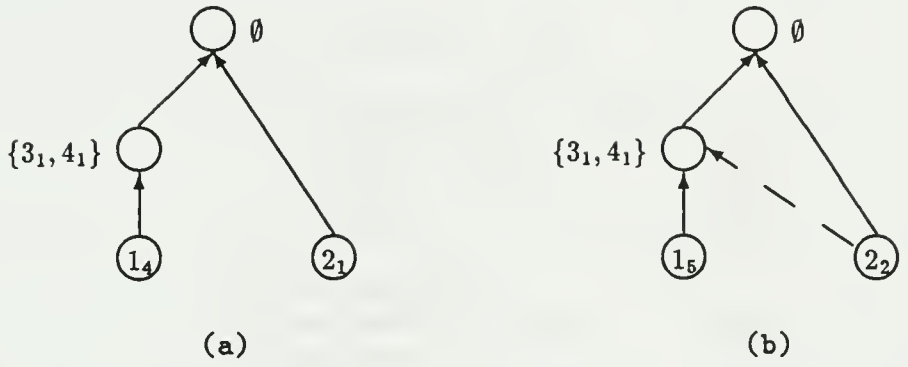


Figure 3.14: Stages in Free Task Dag for POEG in Figure 3.13

dag is now truly a dag rather than restricted to being a tree, as can be seen in Figure 3.14. This means that there may be a choice of nodes when assigning a task to a given block. The goal of on-line task assignment protocols is to select the "correct" node.

A variety of different protocols are possible for assigning tasks. To preserve the MRU heuristic, all assignment protocols are based on a topological traversal of the dag using *time stamps* that are associated with nodes. (The time stamp of a node n is one more than the highest time stamp of the outnodes of n.) A naive task assignment protocol does not differentiate between direct and indirect outnodes when performing the topological traversal; it simply picks the node with the highest time stamp. Unfortunately, the size of the task assignment performed by this protocol is, in the worst case, linear in the number of blocks in the POEG (e.g. $O(B)$). This can be much larger than the bound of T' for the simple MRU algorithm.

Figure 3.15 shows one such worst case assignment. In this very unlucky execution, tasks that are assigned to the left parallel construct at phase $i-1$ are always reassigned to the right parallel construct at phase i. Therefore, new tasks have to be created for the left parallel construct for every phase i. The number of tasks used in the assignment is 10 (approximately $T + \frac{B}{5}$) as compared to an optimal number of 6.

A different assignment protocol bounds the size of the task assignment to T' by giving preference to the tree or direct outnode with the highest time stamp. A task is assigned from an indirect outnode (the one with the highest time stamp) only if the last remaining direct node has an empty free task set. Algorithm 3.11 shows the code for this task assignment protocol.

Theorem 13 proves that in the worst case this protocol will use as many tasks as the simple MRU algorithm.

> **Theorem 13:** The number of tasks in a task assignment performed by the modified MRU algorithm is less than or equal to T'.

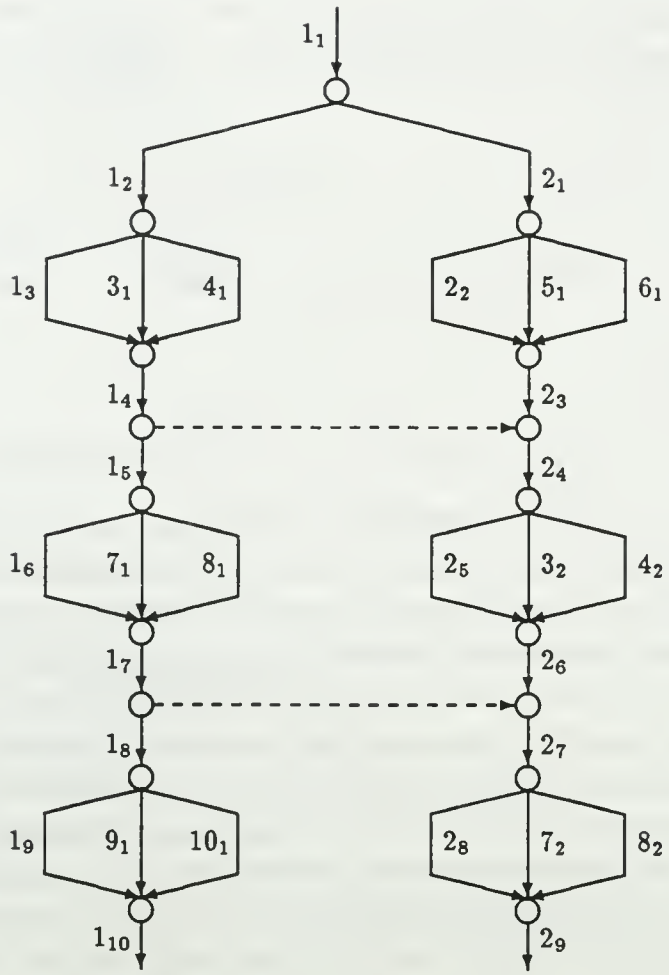


Figure 3.15: Pathological Assignment

However, the modified MRU algorithm should use fewer tasks than the simple MRU algorithm in practice. One open problem in this area is an on-line task assignment algorithm that bounds the size of the assignment to a function of the optimal size.

To illustrate the benefit gained from the modified MRU task assignment algorithm, consider the POEG in Figure 3.14. When block 2_2 performs the fork operation, the children of the fork use tasks 3 and 4 before creating a new task. In this case the modified MRU algorithm performs an optimal assignment for the POEG in Figure 3.13. In contrast, the simple MRU algorithm is forced to generate two new tasks, 5 and 6, to the blocks assigned 3_2 and 4_2 in the optimal assignment. Therefore, more tasks are required for the simple MRU algorithm–six tasks as compared to four–than for modified MRU algorithm.

```
procedure Assign-Task(n_b, task)
    f := n_b
    if f.free-tasks = ∅ then
        f := node in {n.tree ∪ n.direct} with highest time
        if f.free-tasks = ∅ then
            f := node in n.indirect with highest time
        endif
    endif
    if f.free-tasks ≠ ∅ then
        pick task from f.free-tasks
    else task := new task
end procedure
```

Algorithm 3.11: Modified Task Assignment

3.4.4 Complexity of the Modified MRU Algorithm

Unfortunately, a relatively large amount of work is required in the worst case to maintain the extended free task dag. Thus, while the size of its assignment is potentially smaller, the modified MRU algorithm is more expensive than the simple MRU algorithm. This is shown by the following theorems:

> **Theorem 14:** The size of the free task dag used by the modified MRU algorithm is $O((T + C)A)$.
>
> **Theorem 15:** The average cost per block of performing the modified MRU algorithm is $O(A + C)$.

where A is the size of the task assignment (which is bounded by T' by Theorem 13) and C is the number of outstanding asynchronous coordination operations.

These are theoretical worst case upper bounds and are required only for certain pathological POEGs. In general, the size of the free task dag, and hence the cost for task assignment, is much less than these bounds. In fact, the complexity of the free task dag grows as a function of the amount of coordination. For programs with no coordination, the simple and modified MRU algorithms have equal space and time complexities.

Theorem 13 *The number of tasks in a task assignment performed by the modified MRU algorithm is less than or equal to T'.*

Proof. Each block b_j that is assigned from an indirect outnode n can force at most one block b_i to create a new task. However, b_j will use a task from an indirect outnode only if no free task was last assigned to a direct ancestor of b_j. If b_j did not use an indirect outnode, as in the simple MRU algorithm, it would be forced to create a new

task immediately. It follows that the number of tasks used by the modified MRU algorithm is at most the number of tasks used by the simple MRU algorithm which is T'. □

Theorem 14 *The size of the free task dag used by the modified MRU algorithm is* $O((T+C)A)$.

Proof. There can be at most A internal nodes since every such node must have at least one free task associated with it. There are still at most T entry point nodes and C outstanding asynchronous coordination operations. By construction, the internal nodes of the free task dag always form a tree. Therefore, only leaf nodes have more than one out edge. The total number of nodes in the free task dag is $O(A+C)$, the number of edges is $O((T+C)A)$, and hence the total size of the free task tree is bounded by $O((T+C)A)$. □

Theorem 15 *The average cost per block of performing the modified MRU algorithm is* $O(A+C)$.

Proof. As in the proof of Theorem 12, we consider the operations that must be performed for each block: an assignment or coordination and (possibly) a collapse operation when a block is created, and a fork, join or coordination and (possibly) a subsume operation when a block is terminated. The assignment, fork and join operations require $O(A)$ work. The coordination operation also requires $O(A)$ work; a global outnode set is maintained for every coordination point to which every sender block adds its outnodes, and that is copied to each receiver block's entry point after all coordinating blocks have updated it. Since the internal structure of the free task dag is a tree, all collapsed and subsumed nodes n have a single outnode. Therefore, at most $O(T+C)$ work is required to update the tree outnode and direct and indirect outnode lists in which n appears. □

3.5 Extensions to Task Recycling

There are two primary ways in which the techniques developed for the task recycling algorithm can be applied to other issues in debugging parallel programs. One method is to use the set of anomalies reported to increase the reliability of trace-and-replay based debuggers. The second way to extend event-based debuggers to consider both operational and causality ordering constraints when detecting general race conditions.

3.5.1 Trace-and-Replay

Trace-and-replay is one technique of debugging parallel programs. In sequential programs, two execution instances of the same program on the same input vector are guaranteed to be identical. (Recall that we have a very liberal definition of an input vector.) Because of

the presence of race conditions, this property does not hold for parallel programs. The goal of trace-and-replay debuggers is to store enough information about the race conditions in a specific execution instance E_T of a parallel program and input vector pair (P, I) to guarantee that a subsequent re-execution E_R of P on I is identical to E_T. Reproducibility is achieved by maintaining enough information about the execution state when an event e was performed in E_T so that E_T can be reliably reproduced by forcing some set of events to be executed before every e in E_R.

Instant Replay is a trace-and-replay debugger developed by Fowler, LeBlanc and Mellor-Crummey [LMC87,FLMC88]. In Instant Replay, information about the execution instance is recorded in the *process history tape* associated with each process. Every time a coordination operation is performed, a record is added to the history tape of the coordinating process. Process history tapes are used during the replay phase to control the execution of the program. Specifically, an execution instance is reproduced by constraining all coordination operations to occur in the same order in a subsequent re-execution.

The primary drawback of the Instant Replay system is that it considers only race conditions in synchronization and coordination operations. (In other words, those races that lead to POEG nondeterminism.) However, access anomalies are another type of race condition that can affect the execution of the program and hence must be traced and replayed in order to ensure reproducibility.

Reproducibility of access to a shared variable X can be guaranteed by forcing the same set of conflicting accesses to X to appear in both E_T and E_R. However, the technique for monitoring shared variable accesses should be somewhat different from the approach used for coordination operations, since it is undesirable to trace every access to every shared variable when only the execution of access anomalies must be constrained. Instead, information about access anomalies is stored in a *variable history tape* associated with each process. Each process P also has a *step* counter that is incremented every time P accesses a monitored variable. Each monitored variable X has an associated *version number* that is incremented every time X is written.

During the trace execution of the program, E_T, anomalies are detected using the task recycling algorithm. A record–which consists of a step count, version number and auxiliary field–is written to a variable history tape whenever the variable is accessed in an unsafe manner. Because only information about access anomalies is written to the tape, the size of the tape will generally be much smaller than a trace of all accesses to shared variables.

Specifically, when a process P performs a read or write operation that conflicts with a preceding write operation, the version number and current step of P are written to the variable access tape of P. When a process P performs a write operation that conflicts

with a set of preceding read operations, the version number and current step of P as well as the the number of conflicting are recorded in the variable access tape of P. In addition, the version number and step of each conflicting read event is written to the variable history tape of the process that performed the read event. The algorithms for recording the trace information are shown in Algorithm 3.12.

```
procedure Trace-Read-Event(b, X)
    step(b) := step(b) + 1
    if IsConcurrent(b, Writer(X)) then
        write [step(b), version(X), NULL] to tape(b)
    endif
    Subtract-Read(b, X)
end procedure

procedure Trace-Write-Event(b, X)
    step(b) := step(b) + 1
    count := 0
    for all a in Reader(X) do
        if IsConcurrent(b, a) then
            write [step(a), version(X), -1] to tape(a)
            count := count + 1
        endif
    endfor
    if IsConcurrent(b, Writer(X)) or count ≠ 0 then
        [step(b), version(X), count] to tape(b)
    endif
    version(X) := version(X) + 1
    Subtract-Write(b, X)
end procedure
```

Algorithm 3.12: Tracing Read and Write Events

Since the *Trace-Write-Event* routine writes information to the other processes' variable history tapes, the records in a variable history tape can be out of order. Before the replay phase each variable history tape is sorted by the *step* field. A read event that is involved in access anomalies with two write operations will have two identical tape records. In this case, one of the redundant entries is deleted.

During the replay phase, whenever a process P accesses a monitored variable, the current step of P is incremented and compared against the step count at the head of the variable history tape of P. If the current step is less than the tape step, P can continue execution. Otherwise, the current access was involved in an anomaly and must wait for the state of the execution to be the same as in the prior execution.

In particular, P cannot continue execution until the version of X in E_R is the same as the version in E_T. If the current access is a write, the number of conflicting read

operations that happened before the current write operation must also be the same in E_R. This second check guarantees that all concurrent read operations that are supposed to read the previous version of X are completed before the new version of X is written. The algorithms for the replay phase are shown in Algorithm 3.13.

```
procedure Replay-Read-Event(b, X)
begin
    step(b) := step(b) + 1
    if step(b) = step(tape(b)) then
        while version(X) ≠ version(tape(b)) do loop
        read X
        if aux(tape(b)) = -1 then count(X) := count(X) + 1
        move to next square on tape(b)
    else read X
end procedure

procedure Replay-Write-Event(b, X)
begin
    step(b) := step(b) + 1
    if step(b) = step(tape(b)) then
        while version(X) ≠ version(tape(b)) and count(X) ≠ aux(tape(b)) do loop
        write X
        count(X) := 0
        version(X) := version(X) + 1
        move to next square on tape(b)
    else write X
end procedure
```

Algorithm 3.13: Replay Read and Write Events

Theorem 16 proves that this system is sufficient for tracing and replaying accesses to shared variables.

> **Theorem 16:** Every read event gets the same value in trace execution E_T and replay execution E_R.

The amount of replay control data and overhead is proportional to the number of access anomalies, rather than the number of accesses to shared variables. In addition, the step count must be incremented for every access to every shared variable that has at least one access anomaly.

Lemma 7 *All write events happen in the same order in trace execution E_T and replay execution E_R.*

Proof. Suppose, for the sake of contradiction, that write event w_i to shared variable X happened before write w_j to X in E_T and after w_j in E_R. Let $W = w_i, w_{i+1}, \ldots w_j$ be

the sequence of write operations that occurred in E_T. Consider a write operation $w_k \in W$. If w_{k+1} was performed by a descendant of w_k in E_T, then w_{k+1} must also happen after w_k in E_R. If w_k was concurrent with w_{k+1} in E_T, then there is a record in the variable history tape of the process executing w_{k+1} that forces it to execute after w_k in E_R. Therefore, for all $i \leq k < j$, write w_{k+1} must happen after w_k in E_R, and hence, w_j must happen after w_i in E_R. □

Lemma 8 *All read events happen after the same write events in trace execution E_T and replay execution E_R.*

Proof. Suppose, for the sake of contradiction, that a read event r of shared variable X happened immediately after write event w_i of X (e.g. before w_{i+1}) in E_T, but before w_i in E_R. Read event r must be concurrent with w_i. Therefore r was detected as an access anomaly with respect to w_i in E_T and there is a record in the variable access tape of the process performing r that forces r to execute only after w_i is performed. □

Lemma 9 *All read events happen before the same write events in trace execution E_T and replay execution E_R.*

Proof. Suppose, for the sake of contradiction, that a read event r of shared variable X happened immediately before w_i of X (e.g. after w_{i-1}) in E_T, but after w_i in E_R. Read event r must be concurrent with w_i. By Lemma 8, no read event that happened after w_i in E_T will happen before w_i in E_R. The last read event r' performed by a descendant of r that occurred before w_i was detected as an access anomaly with respect to w_i in E_T. Therefore, r' is guaranteed to happen before w_i in E_R. Otherwise, the *count* field will not be large enough to let w_i proceed. Thus, w_i will not continue in E_R until r'–and therefore its ancestor r if $r \neq r'$–has executed. □

Theorem 16 *Every read event gets the same value in trace execution E_T and replay execution E_R.*

Proof. This follows directly from Lemma 7, Lemma 8 and Lemma 9. □

3.5.2 General Race Conditions

Event based debugging is a technique for detecting race conditions of general events in parallel and distributed programs. Event-based debugging was first proposed by Bruegge [Bru85,BH83] and has been used as the basis of many later debugging systems [Bat88,BW83,HK88,Sto88]. (Access anomalies are a very specific type of race condition in which the only possible events are read and write operations.) To detect general race conditions, the set of events to monitor and the notion of conflicting events must be explicitly

defined by the programmer. (In access anomaly detection systems, the set of events and notion of conflicting events are defined by Bernstein's Conditions.)

Most event-based debugging systems use mechanisms based on *path expressions*–a protocol proposed by Campbell and Habermann [CH73,Hab75]–for specifying the valid ordering of events. The syntax and semantics of path expressions are shown in Figure 3.16. An execution instance is valid with respect to a path expression P if and only if it

pe := event		
| $pe_1 + pe_2$	selection	- either pe_1 or pe_2 may execute, but not both
| $pe_1 ; pe_2$	sequencing	- pe_1 must execute before pe_2
| pe^*	repetition	- pe may execute zero or more times

Figure 3.16: Syntax and Semantics of Path Expressions

corresponds to a string in the language defined by P. Event-based debuggers use a set of path expressions to fully specify the desired ordering constraints.

Event-based debuggers guarantee that the following operational constraint on the execution order of events is maintained during the execution of a parallel program:

> **Operational Constraint:** Event e is valid if and only if e is a valid next symbol with respect to *all* path expressions in which e appears.

In other words, they check that the sequence of operations performed is valid with respect to all path expression constraints. A weakness of event based debugging systems is that no attempt is made to verify that the operational constraint is enforced by the semantics of the parallel program[6].

To illustrate this problem, consider a parallel program in which there is a producer and a consumer with a one-unit buffer. The consumer may proceed only after something has been produced, and the producer may proceed only if its previous output has been consumed. This behavior is checked by the path expression below:

path *produce*; *consume* end

where path ... end is a repetition operator. The following execution sequence,

produce, *consume*, *produce*, *consume*, ···

meets the operational constraints imposed by the above path expression. Therefore, this execution would be considered valid by an event-based debugging system. If the first *produce* event was not causally ordered with the *consume* event, however, a race condition

[6]The system of Hseush and Kaiser addresses this problem by requiring exact ordering relationships [HK88,HK90].

exists which was not detected because the two events simply happened to occur in the correct order.

The notions used in task recycling can be encorporated into event-based debugging systems to improve the race condition detection. Specifically, access histories and a POEG are used to verify that the following causality constraint is also met.

> **Causality Constraint:** Event e must not be concurrent with any event that is required to precede event e by the semantics of the path expressions.

Thus, race conditions will be detected regardless of whether or not the events happen to be performed in the correct order in a given execution instance. By requiring that both operational and causality constraints are met, race conditions detection is strengthened in these systems.

The approach of checking causality ordering is fairly independent of the mechanism for specifying and checking operational ordering constraints. Whatever rules are applied to verify operational ordering can be modified to verify causality ordering as long as we can associate an access history with each operational ordering constraint. The remainder of this section describes the algorithm for checking causality ordering for path expressions.

Because a path expression is a regular expression, it has a corresponding deterministic finite automaton. Each event in a path expression is represented by a transition and each ";" sequencing operator by a state. A state s becomes *selected* when an event associated with an in-transition of s executes; s becomes *unselected* when an event associated with an out-transition of s executes. All of the out-transitions of the selected state are *permissible*. An event e can execute if and only if there is a permissible transition labeled e in all of the path expressions in which e appears.

As in access anomalies detection, access histories contain the identifiers of blocks and are checked and unneeded entries subtracted whenever an event is performed. In event-based debugging, the access history is distributed by associating entries with states in the automata. The access history entry of state S contains the identifier of the block which last selected S. When a block b executes an event e, block b checks if it is concurrent with any block in the access history entry associated with the head states of each of its permissible transitions. If so, a race condition exists. Block b then adds itself to the tail states' access histories. Algorithm 3.14 shows the algorithm for verifying valid operation ordering and checking causality constraints.

```
procedure Check-Event(e, b)
    if | permissible(e) | < paths-in(e) then
        report Operational Constraint Error
    else
        for all t in permissible-trans(e) do
            if IsConcurrent(b, access-history(t.head)) then
                report Causality Constraint Error
            endif
            for all transitions s in out-trans(t.head) do
                delete s from permissible(s.event)
            endfor
            for all transitions s in out-trans(t.tail) do
                add s to permissible(s.event)
            endfor
            access-history(t.tail) := b
        end for
    end if
end procedure
```

Algorithm 3.14: Check Operational and Causality Constraints

Chapter 4

Empirical Measurements of Task Recycling

Chapter 3 presented the task recycling algorithm for detecting access anomalies in parallel programs. We can analytically parameterize the overhead associated with task recycling. For instance, the cost incurred whenever a monitored variable X is written is proportional to the number of concurrent blocks that have read X; in the worst case, this cost is equal to the maximum concurrency of the program. Similar bounds exist for maintaining concurrency information. General discussions of algorithmic overhead, however, give little insight into the actual costs incurred when detecting anomalies in parallel programs.

To better evaluate the task recycling algorithm–as well as the general approach on-the-fly detection of access anomalies–the task recycling algorithm was implemented for a parallel Fortran environment on the NYU Ultracomputer. In order to obtain relative performance measurements, an alternative on-the-fly technique known as *English-Hebrew labeling* [NR88a,NR88b] was also implemented. Implementation of these two algorithms is discussed in Section 4.1.

Measurements were gathered for each of the following benchmark parallel programs:

Triso - Solves a sparse triangular linear system equations using wavefront parallelism

Finite - Solves a linear system using finite element methods

Simple - Solves partial differential equations for hydrodynamics and heat conduction

Polymer - Performs molecular dynamic calculations of polymer systems

Section 4.2 presents information about the parallel structure and access patterns of the benchmark programs. The behavior seen in the benchmark programs justifies design decisions made in the development of the task recycling approach and inspired several optimizations.

Section 4.3 discusses the overhead (measured in space, elapsed running time and CPU execution time) for monitoring parallel programs using the task recycling and English Hebrew labeling techniques.

The experimental data indicate four important results:

1. The benchmark programs use data partitioning so extensively that over 80% of all variables never have more than two concurrent readers. Therefore, the size of an access history is generally very small and independent of the degree of parallelism of the program.

2. For the benchmark programs, monitoring entails a 150% to 350% slowdown. Although this cost is high, it is not unreasonable during the debugging phase of program development.

3. Because of its efficient concurrency information management, English-Hebrew labeling performs substantially better than unoptimized task recycling on programs with frequent parallel operations and relatively few accesses to shared variable. However, optimizations can be used to reduce the costs incurred by task recycling to be proportional to the complexity of the concurrency structure and thus, comparable to the cost incurred by English-Hebrew labeling.

4. As reader sets or concurrency lists increase in size, the high cost of performing concurrency verification in English-Hebrew labeling outweighs the benefits of its efficient concurrency information maintenance.

If the benchmark programs are indicative of a wide class of parallel programs, the task recycling algorithm is an important improvement over existing technology.

4.1 Implementation

Task recycling and English-Hebrew labeling have been implemented for the parallel Fortran environment on the NYU Ultracomputer [Ber88,Got88]. The following sections describes the system for instrumenting parallel programs and the implementation details for the English-Hebrew labeling and task recycling techniques. Because of the fairly simple concurrency structure of the benchmark Fortran programs, several additional implementation optimizations were possible and are described below.

4.1.1 Instrumentation System

The version of parallel Fortran available on the Ultracomputer is based on Fortran-77. It supports doall and parallel case constructs for creating parallel threads; barriers and

```
PROGRAM EXMPLE
SHARED /GLOBAL/ A(4), X
INTEGER*4 I, J
X = 0
DOALL I = 1,3,1
    A(I) = I * I
    J = X + A(I+1)
ENDALL
END
```

Figure 4.1: Example of Fortran Code

critical sections are the primary forms of coordination. Figure 4.1 shows a simple parallel Fortran program. All shared variables are explicitly defined by specifying a common block to be SHARED. The DOALL construct creates three concurrent iterates.

A simple front-end preprocessor is used to instrument a parallel program. The preprocessor approach makes the system relatively portable to other parallel Fortran system and simplified the implementation. However, the resulting monitoring efficiency could be improved. A more complex preprocessor or compiler-based front-end can use static analysis or user directives to reduce the monitoring cost in several ways. First, only those accesses flagged as potential access anomalies by static analysis need to be monitored during run-time. Second, monitoring code could be moved out of a loop body as long as there are no thread creation, termination or coordination operations within the loop. In addition, concurrency information has to be maintained only in portions of the program where variable access monitoring code is present. In this implementation, every access to every shared variable and every parallel operation is monitored.

The preprocessor allocates storage for access histories and concurrency state information. Every monitored variable has a "mirror" variable that contains its access history. The name of the access history is simply the name of the variable suffixed by a "Thus, when a block reads variable $A[i]$, the access history in $A\%[i]$ is checked for anomalies and then updated. Each entry in an access history contains a task identifier or a pointer to the English-Hebrew label. It also contains the line number and a pointer to the function name where the last access was performed. This information enables us to identify the locations in the program code where access anomalies occur and produce useful error messages.

Due to memory constraints on the Ultracomputer, the reader sets in the access histories are of fixed length. In particular, the benchmark programs were monitored using reader sets with one and two entries. While this limitation may result in undetected anomalies, results discussed in Section 4.2.2 indicate that anomalies will be missed only rarely. In fact, for half of the benchmark programs a reader set of size two is sufficient to guarantee

that an anomaly is detected for every variable that is accessed in an unsafe manner.

The preprocessor inserts calls to library routines for maintaining access histories whenever a shared variable is accessed and for updating concurrency information at every thread creation and termination point. The routines for checking access histories and performing subtraction on access histories are written in assembler for reasons of efficiency. The task recycling and English Hebrew labeling access anomaly detection algorithms are implemented by two library packages written in C. The instrumented program is linked to a separate library of instrumented coordination routines–namely, barrier and critical section coordination–that update concurrency information at coordination points.

Figure 4.2 shows the code generated by the preprocessor for the program in Figure 4.1. The programmer uses a `MONITOR` directive to specify which shared variables to monitor.

```
PROGRAM EXMPLE
SHARED /GLOBAL/ A(4), X
shared /global%/ A%(9,4), X%(9)
INTEGER*4 I
integer*4 A%,X%
common/myfunct/ myfun
character*16 myfun
call initds()
call setstr(myfun, 'exmple')
call writxx(X%,5)
X = 0
call mnfrkp()
DOALL I = 1,3,1
    call mnfrkc()
    call writxx(A(1,i)%,7)
    A(I) = I × I
    call readxx(X%,8)
    call readxx(A(1,i+1)%,8)
    J = X + A(I+1)
    call mnjonc()
ENDALL
call mnjonp()
END
```

Figure 4.2: Example of Instrumented Fortran Code

In Figure 4.2, the `GLOBAL` common block was monitored. Some of the inefficiencies of the preprocessor approach can be seen in that variable `X` is monitored during a portion of sequential execution. This unnecessary code would not be added if a front-end system were used to first locate potential access anomalies.

The following is the anomaly report for one possible execution instance of the program

in Figure 4.2 :

```
Write-Read anomaly for A(3):  in exmple.7 and exmple.8
Read-Write anomaly for A(2):  in exmple.8 and exmple.7
```

In this execution instance, iterate 2 completed before iterates 1 and 3 began executing. Iterate 2 added a read event to A%(3) and a write event to A%(2). When iterate 3 executed, it detected that its write of A(3) on line 7 conflicted with a prior read of A(3). When iterate 1 executed, it detected that its read of A(2) on line 8 conflicted with a prior write.

4.1.2 English Hebrew labeling

English-Hebrew labeling is an on-the-fly anomaly detection scheme developed by Nudler and Rudolph [NR88a,NR88b]. In English-Hebrew labeling, the direct ancestor relationship–that is the partial ordering introduced by parallel constructs–is encoded in a *tag* associated with each block. A tag consists of a pair of labels: an *English label E* and a *Hebrew label H*. Conceptually, the English label is produced by performing a left-to-right preorder numbering of the POEG; each block is assigned a number less than all of the numbers assigned to its children and its siblings to the right. The Hebrew label is produced by performing a right-to-left numbering.

Since the labels must be generated on-line, a complete traversal of the POEG cannot be performed. Therefore, a label is a string of numbers and labels are *lexicographically* ordered. The children blocks $c_0 \ldots c_m$ of a fork vertex with parent block p are assigned English and Hebrew labels as follows:

fork: $E(tag(c_i)) := E(tag(p)) \mid i$
$H(tag(c_i)) := H(tag(p)) \mid m - i + 1$

where | is the append operation and i is a unique child number. The child block c of a join vertex with parents $p_0 \ldots p_m$ is assigned the English and Hebrew labels:

join: $E(tag(c)) := max(E(tag(p_i)))$
$H(tag(c)) := max(H(tag(p_i)))$

A block c created in a coordination operation that has parent p is assigned English and Hebrew labels:

coordination: $E(tag(c)) := E(tag(p)) \mid 1$
$H(tag(c)) := H(tag(p)) \mid 1$

As specified above, the length of the labels increase with the number of fork and coordination operations. However, an additional heuristic described in [NR88a,NR88b] bounds the label length to the level of nesting, $O(N)$.

Two tags t_i and t_j are *unordered* if and only if the following condition is met:

$$(E(t_i) < E(t_j) \text{ and } H(t_i) > H(t_j)) \text{ or } (E(t_i) > E(t_j) \text{ and } H(t_i) < H(t_j))$$

The tags of a block and any of its direct ancestors or descendants are always ordered. However, English-Hebrew tags only encode concurrency properties for the fork and join operations. When two blocks are not concurrent in the POEG because of explicit coordination, their tags are unordered. *Coordination lists* are used to record execution orderings imposed by coordination.

Specifically, a coordination list is associated with each executing block b to store the tags of the indirect ancestors of b. All tags in a coordination list are unordered, so that the length C of a coordination list is bounded by $O(T)$. The test for concurrency between a block b and tag t requires determining if $tag(b)$ or any of the tags in the coordination list of b are ordered with t. Algorithm 4.1 shows the code for detecting if block b and tag t are concurrent.

```
procedure IsConcurrent(b, t)
    if E(tag(b)) > E(t) and H(tag(b)) > H(t) then return false
    else
        for all tags c is coordination-list(b) do
            if E(c) > E(t) and H(c) > H(t) then return false
        end for
    end if
    return true
end procedure
```

Algorithm 4.1: English-Hebrew Concurrency Check

Figure 4.3 illustrates the use of English-Hebrew labels for a POEG. The coordination list of block **1131,1231** contains the tag **121,113** because **121,113** is an indirect ancestor of **1131,1231**. Likewise, the coordination lists of block **121,113** contains the tag **113,123**. We can determine that blocks **1131,1231** and **11,12** are not concurrent because their tags are ordered; namely, **1131** > **11** and **1231** > **12**. Similarly, blocks **1131,1231** and **12,11** are not concurrent because a tag in the coordination list of **1131,1231**–namely, **121,113**–is ordered with **12,11**. However, blocks **1131,1231** and **123,111** are concurrent because neither the tag **1131,1231** nor any entry in its associated coordination list is ordered with **123,111**.

Coordination lists are similar to parent vectors in that it is necessary to keep coordination lists only for the currently executing blocks. New coordination lists are formed by merging the coordination lists of all parents and adding indirect parents. Comparing any two English-Hebrew labels requires $O(N)$ work. Because the number of English-Hebrew tags is unbounded, the coordination list cannot be directly indexed as an array. Hence, the cost of checking for a conflict at each variable access is bounded by $O(N \times C)$. This problem can be addressed by maintaining the English-Hebrew labels in both the concur-

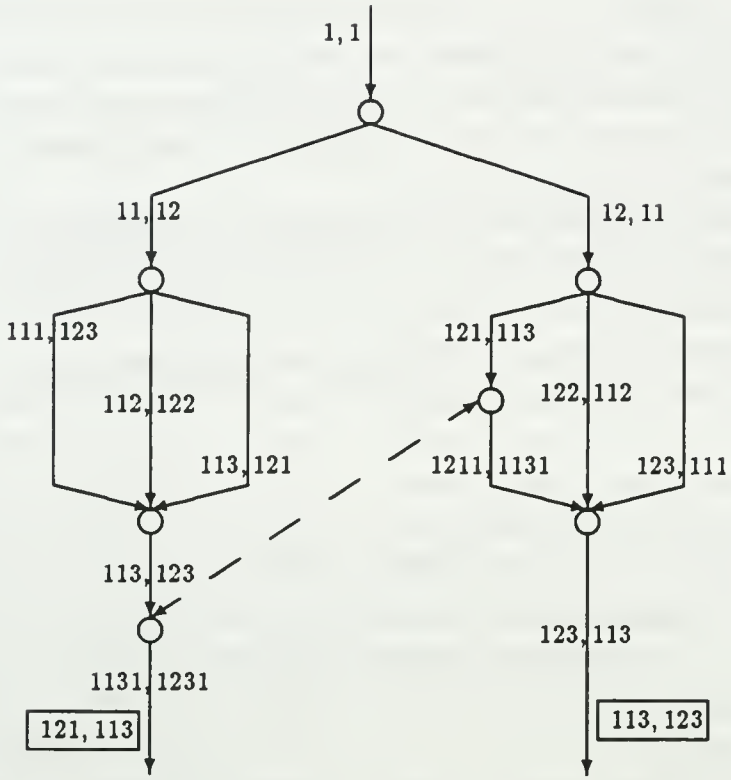


Figure 4.3: English-Hebrew Labeling

rency lists and in the access histories in sorted order, so that standard binary search and sort-merge techniques can be applied.

An implementation difficulty with English-Hebrew labeling is managing the tags. Since English and Hebrew labels are of variable length, storage allocation becomes an issue. If tags are stored directly in access histories, the amount of storage needed is significantly increased, since the number of access histories is $O(V)$, and it is not unreasonable to assume V to be of the order of several million. If the tags are stored indirectly, there is a garbage collection issue. In this implementation, English-Hebrew labels are stored indirectly in access histories to decrease access history size. The generation count optimization (described below) is used to perform garbage collection.

4.1.3 Task Recycling

The implementation of the task recycling technique uses modification sets to maintain parent vectors and the simple MRU algorithm to perform task assignment. Because most parallel scientific Fortran programs do not have a complex concurrency structure, the

simple MRU algorithm generally performs well for this class of programs. In fact, it is guaranteed to be optimal for all of the benchmark programs.

In order to minimize contention in assigning and freeing tasks, several free task sets are associated with each node in the free task dag. When a block needs to be assigned a task from node n, one of the free task sets associated with n is selected based on a "random" value (e.g. the processor id). If the number of free task sets associated with a node is similar to the underlying parallelism, the contention for the free task dag is greatly reduced, although more tasks may be needed. In our implementation, the Fetch&Add operation is used to select a free task set, and the number of free task sets is equal to the underlying parallelism of eight.

4.1.4 Generation Counts

A *generation count* is a mechanism for grouping blocks that are potentially concurrent. The generation is incremented every time the execution of the program is decreased to a single thread or increased to more than one thread. Two blocks that are not in the same generation are guaranteed not to be concurrent. However, two blocks in the same generation are not necessarily concurrent. Further analysis is required to determine the concurrency relationship between blocks in the same generation.

For instance, there are seven generations in the POEG in Figure 4.4. All blocks in an outermost parallel construct are in same generation. No block in generation 4 is concurrent with any block in preceding generations 1, 2 or 3 or subsequent generations 5, 6 or 7. However, not all blocks in generation 4 are concurrent.

Generation counts are used by both task recycling and English Hebrew labeling to significantly reduce the space and computation time requirements. Since many scientific Fortran programs consist of long series of generations, similar benefits may be achieved for many programs in this class. (In contrast, many Ada program are a single generation, since concurrent tasks can be created when a program begins which terminate only when the program completes. For this class of programs, the generation count will only increase access history sizes without decreasing the overall computation time or space.)

To use generation counts, the current generation count is stored with each entry in an access history. When a block checks an entry in an access history, it first checks the generation count. If the generation count of the entry is less than the current generation count, the block cannot be concurrent with the entry and therefore does not need to check its parent vector (in the case of task recycling) or compare tags (in the case of English Hebrew labeling). A concurrency check is performed only if a variable is accessed by two blocks in the same generation.

For the English-Hebrew labeling technique, all tags associated with blocks in earlier

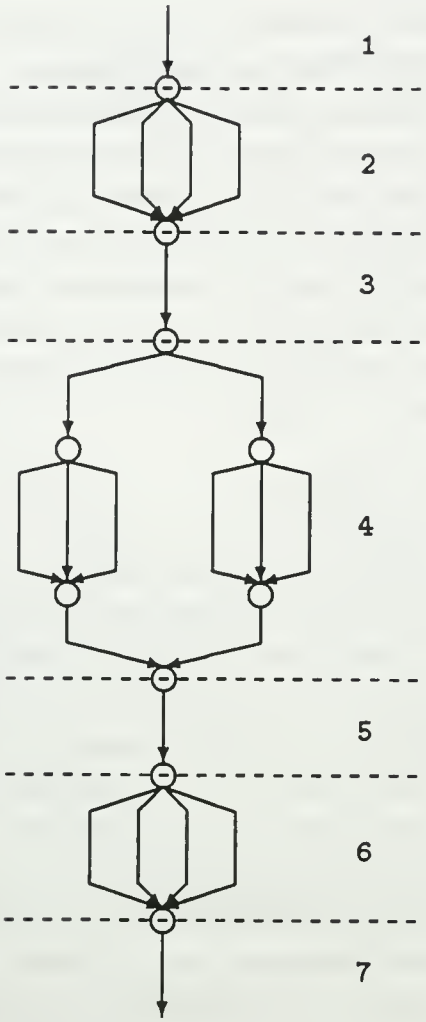


Figure 4.4: POEG with Generations

generations can be discarded when a new generation is entered. This method for performing garbage collection can result in dramatic space savings. In addition, the number of expensive tag comparisons is decreased. For task recycling, generation counts decrease the amount of work performed maintaining parent vectors. In particular, parent vectors need to accurately represent ancestor relationships only among blocks in the same generation. Therefore, no parent vector maintenance is needed in programs with no nested parallelism or coordination.

4.1.5 Parallel Fortran Environment

The "run-until-completed" scheduling paradigm is the method used in the Ultracomputer parallel Fortran and in many other parallel Fortran environments. In the run-until-completed model, a thread created in a doall will not block until it terminates at the associated endall operation. Although parallel scientific codes typically exhibit a high degree of *nominal* parallelism, at any point in time the number of threads created but not terminated is limited to P, the underlying parallelism of the machine. Therefore, at most P parent vectors or coordination lists are actually associated with currently executing threads at any point during the execution.

In the parallel Fortran environment, each of P actual processes perform the work of several parallel threads, and the differences among the private parent vectors of these threads are very small. Task recycling takes advantage of this property to reduce the work performed in initializing parent vectors in programs with strictly nested parallel constructs (e.g. no nested series of parallelism).

A private stack of N "template" parent vectors is cached where N is the level of parallelism nesting. After a block terminates, the modification set associated with its current parent vector is used to update the new parent vector as described in Section 3.2. The modification set is also used to remove all changes made to the old parent vector p, thereby setting p back to its initial state. When the next parallel thread is executed, a new parent vector does not have to be allocated and re-initialized. The number of times a change is either added or backed out of a parent vector is equal to the nesting level of the block that is associated with the change. Thus, the cost per block for maintaining parent vectors is reduced from $O(T)$ to $O(N)$ and the amount of work per process is $O(T \times N)$.

4.2 Program Behavior Results

The first goal in performing these experiments is to gain insight into the programming style found in "real" parallel programs. In particular, we were interested in gathering data about their concurrency structure and shared variable access patterns. Information about the concurrency structure allows us to estimate average values for the maximum concurrency, T, and total number of blocks, B. It also indicates the viability of various optimization techniques and the expected benefit of the modified MRU task assignment algorithm. Shared variables access patterns influence the size and work required for maintaining access histories. Our findings on parallel program behavior are presented below.

4.2.1 Concurrency Structure

The concurrency structure of all four benchmark parallel programs is quite simple: there is very limited nesting of doall constructs and minimal synchronization. This means that the optimizations described in Sections 3.2 and 4.1 significantly improve the performance of task recycling and English-Hebrew labeling. In addition, the simple MRU algorithm is guaranteed to perform an optimal task assignment for all of the benchmark programs.

However, the degree and granularity of parallelism varies considerably among the benchmark programs.

- Triso has coarse granularity parallelism with limited synchronization. It consists of a single doall operation which creates 8 parallel threads; these threads subsequently perform two barrier synchronization operations.

- Simple has medium granularity parallelism with some coordination. It performs 10 doall operations that each create 124 parallel threads, and 130 doall operations that create from 10 to 30 threads. In addition, during 10 phases of execution approximately 15 concurrent threads execute a critical section.

- Finite exhibits a large degree of fine granularity parallelism and does not perform any coordination. It performs 60 doall operations that each create 1000 parallel threads; 50 doall operations that create 250 threads, and 200 doall operations that create between 2 and 32 threads. Each block performs a very limited amount of computation; in many cases, a block consists of a single operation on an array element.

- Polymer exhibits a large degree of medium granularity parallelism and does not coordinate. It has one level of nested parallelism; the first three benchmark programs do not have nested parallelism. It performs 40 nested doall operations; the outer operations create 1000 parallel threads each of which creates 3 parallel sub-threads. In addition, it performs 20 doall operations which create 350 parallel threads and 10 doall operations which create 100 parallel threads.

Table 4.1 summaries the concurrency parameters for the benchmark programs.

The experimental results presented in Section 4.3 show that the concurrency structure of the program has a significant impact on the cost of maintaining concurrency information. For instance, the unoptimized version of task recycling performs very poorly on programs with fine granularity parallelism, such as Finite. Surprisingly, the concurrency structure does not significantly impact the cost per variable access, as is discussed in the following section.

Program	*Total # of Blocks*	*Maximum Concurrency*
Triso	24	8
Simple	4,090	124
Finite	74,900	1,000
Polymer	128,000	3,000

Table 4.1: Concurrency Structure

4.2.2 Shared Variable Access Patterns

The work and space required for maintaining access histories is proportional to the average size of the reader set. While theoretically the size of the reader set may grow to the maximum concurrency of the POEG, in practice the number of concurrent readers is much smaller. This is due to a common technique for designing parallel scientific programs. Many parallel scientific programs distribute the workload by partitioning data among concurrent threads. Hence a thread often shares data with one or two neighboring threads, but seldom shares data with all other threads. Thus, one would expect the number of concurrent readers to be limited in programs based on this "data partitioning" paradigm.

The shared memory access patterns measured for the benchmark programs support this conclusion and are shown in Figure 4.5. The number of concurrent readers goes from 1 to 9 along the x-axis, and the percentage of accessed shared variables that are read by at most R concurrent blocks goes from 45% to 100% along the y-axis. (Shared variables that are never accessed are not included in these statistics.) For example, 95% of the shared variables in the Triso program are never read by more than two concurrent readers at any point during execution.

The most significant result of these measurements is that the number of concurrent readers tends to be very small. In all four programs, more than 80% of the variables are never read by more than two concurrent readers and almost 50% are never read concurrently. In addition, there appears to be little correlation between the number of concurrent readers and the degree of parallelism of the program. For example, the Triso and Polymer programs have fairly similar access patterns for the majority of their variables (excepting the 9.3% in the Polymer program with more than 9 concurrent readers). However, Triso has modest parallelism while Polymer has nested parallelism of a very high degree.

Table 4.2 shows estimates of the average reader set size[1]. In fact, these figures are

[1]Any variable with more than 9 concurrent readers was assumed to have the maximum number of concurrent readers. For the first three benchmark programs, this does not affect the estimated average reader set size. However, 10% of the variables in the Polymer program have more than 9 concurrent readers. In this case we are slightly less conservative and assume that these variables are accessed by the *average* number of concurrent blocks.

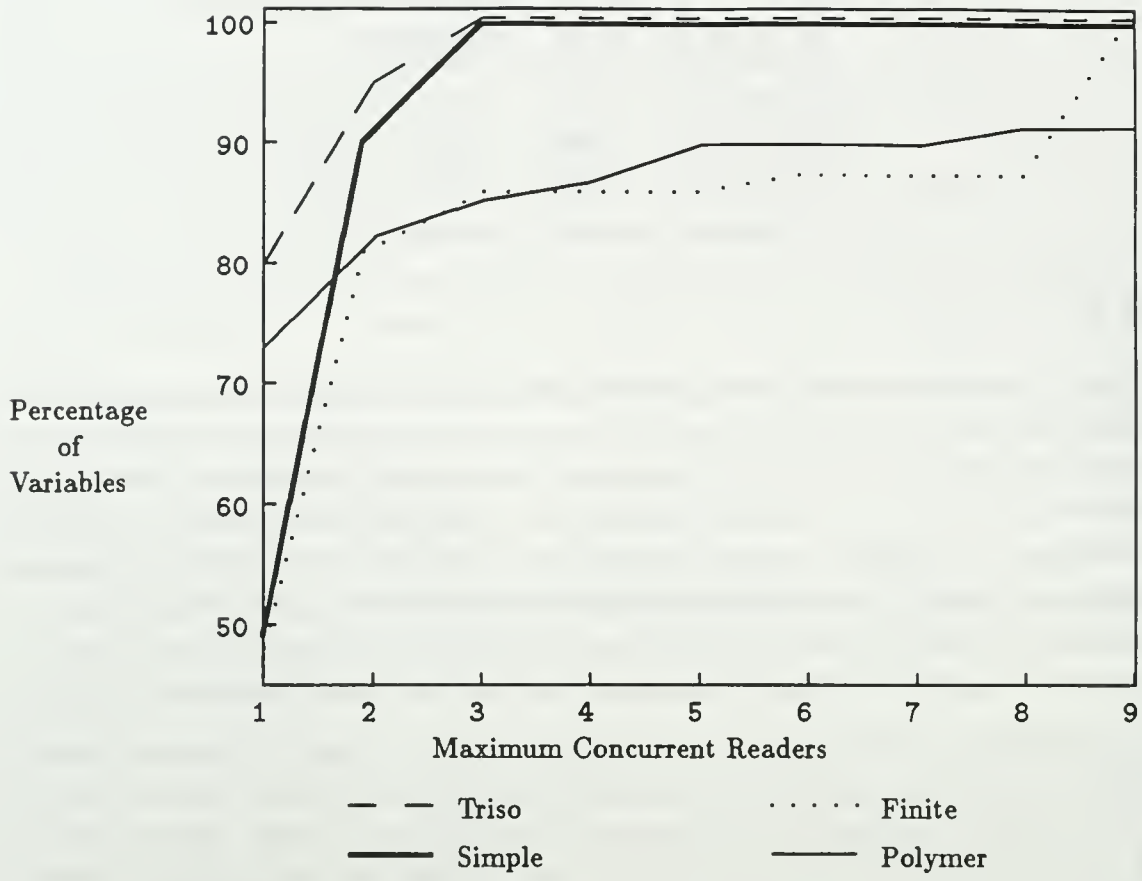


Figure 4.5: Number of Concurrent Readers

pessimistic with respect to the average reader set size, since they assume that a variable that is accessed by at at most n concurrent blocks at some point during the execution always has n entries in its reader set. A variable with a maximum number of concurrent readers of n may actually have a much smaller reader set size throughout most of the execution of the program.

A direct consequence of the limited reader set sizes is that the estimated number of shared variables that are read per block is much smaller than the total number of variables. (If a program has V shared variables, average concurrency of T_{ave}, and average reader set size of R, the average number of variables read per block is computed as $\frac{V \times R}{T_{ave}}$.) Since the average number of variables read by a block–as well as the average number of concurrent blocks that read a given shared variable–is very small, the access history based algorithms are preferable to the approach used by the Merge algorithm [Sch89,Sni88] in which the storage per variable depends only on the maximum concurrency of the POEG and not on

Program	*Ave. Reader Set Size*	*% Vars Read Per Block*
Triso	1.25	23%
Simple	1.69	7%
Finite	2.71	1%
Polymer	168	9%

Table 4.2: Estimated Average Reader Set Sizes and Variables Accessed Per Block

the number of accessing blocks.

4.3 Monitoring Results

The second goal in performing these benchmark measurements is to understand the actual performance impact of on-the-fly access anomaly detection on parallel programs. To this end, the elapsed running time, CPU time and space requirements for the task recycling technique were measured and compared with the costs incurred by English-Hebrew labeling. The primary difference between the two algorithms is that the cost per variable access is clearly smaller in task recycling than in English-Hebrew labeling. Because variable access is frequently the most common operation, this is an important attribute of task recycling. However, concurrency information management is potentially much less expensive for English-Hebrew labeling. The benchmark results provide some insights into the significance of the tradeoffs between the two algorithms. These measurements also enable us to evaluate the effectiveness of various optimization strategies.

Four versions of each program were executed:

Unmonitor - unmonitored program

Concurrency - maintains concurrency information but accesses to shared variables are not monitored

Monitor(1) - monitors every shared variable using a reader set size of one

Monitor(2) - monitors every shared variable using a reader set size of two

These versions isolate the costs of maintaining concurrency information and updating access histories.

4.3.1 Space Requirements

Figure 4.3 compares the storage requirements for maintaining concurrency information in task recycling and English-Hebrew labeling. As is shown in Figure 4.3, English-Hebrew labeling requires substantially more space than task recycling when the generation count optimization is not used. For example, almost a megabyte of memory is required to

Program	Task Recycling	English-Hebrew	
		Unopt	Opt
Triso	2	8	2
Simple	6	80	7
Finite	29	954	20
Polymer	279	7,000†	239

†Estimated Value

Table 4.3: Concurrency Information Space Requirements (in Kbytes)

monitor the Finite program with the English-Hebrew labeling technique whereas the task recycling technique uses only 29 kilobytes. In fact, English-Hebrew labeling could not monitor the Polymer program due to memory constraints on the Ultracomputer.

When generation counts are used, however, task recycling and English-Hebrew labeling require approximately the same amount of space for maintaining concurrency information. (All English-Hebrew labels are discarded at the end of every generation.) Therefore, the generation count optimization is very important since it broadens the class of programs supported by the English-Hebrew labeling technique. For programs with infrequent serialization points, the excessive space requirements of English-Hebrew labeling may make its use infeasible.

4.3.2 CPU Times

The user mode CPU time of a parallel program P is the sum of the execution times of all of the threads in P, and it reflects the total amount of work performed by P. Figure 4.6 shows the user mode CPU time for executing the *Concurrency* version of each benchmark program with both the optimized and unoptimized versions of the anomaly detection algorithms. (*Concurrency* isolates the cost of maintaining the concurrency information from the overall cost of detecting access anomalies.) The times are shown as a percentage of the CPU time for the *Unmonitor* version, and are computed as:

$$\frac{Concurrency \times 100}{Unmonitored}$$

For example, there is 175% increase in CPU time to maintain concurrency information for the Polymer program using the unoptimized version of the task recycling technique.

As can be seen, cost of maintaining concurrency information can be substantial for programs with very high degrees of fine-grained parallelism. For example, the overhead incurred by unoptimized task recycling is twice as much as English-Hebrew labeling for Simple and Finite and up to six times as much work is performed as executing the original versions of the programs. (The unoptimized version of English-Hebrew labeling could not be used to monitor Polymer due to memory limitations.)

One can also compare the cost of maintaining concurrency information both with and without optimizations in place. When the generation count optimization is used, task recycling performs as well or better than English-Hebrew labeling for all programs. Thus, the generation count optimization leads to a decrease in the computation cost for task recycling that is as striking as the decrease in space for English-Hebrew labeling.

Figure 4.7 displays the increase in user mode CPU time when monitoring access to shared variables. In particular, the *Concurrency*, *Monitor(1)* and *Monitor(2)* versions of each program were executed using the optimized anomaly detection algorithms. The most important result shown in Figure 4.7 is that English-Hebrew labeling requires more time than task recycling for monitoring variables, even when the cost of maintaining concurrency information is comparable (for example, Polymer).

A comparison of the execution times for *Monitor(1)* and *Monitor(2)* in Figure 4.7 shows that reader set sizes can be increased from one entry to two entries with little additional cost for task recycling (less than 30% for all programs). The percentage of variables guaranteed to have at least one anomaly detected is increased by 60% (from 50% to 80%). As reader sets grow in size, the cost of the concurrency check becomes more important. Therefore, the overhead grows more rapidly for English-Hebrew labeling than for task recycling.

4.3.3 Elapsed Running Times

An alternative metric of the computation overhead of executing a parallel program is the *elapsed running time* of the execution. This reflects the overall cost for executing the program in a stand-alone environment and perhaps is the more interesting measurement for users of an anomaly detection system. Figure 4.8 shows the increase in total elapsed times for the *Concurrency* and *Monitor(2)* versions of each of the benchmark programs using the optimized anomaly detection algorithms.

Maintaining concurrency information results in a 2% - 78% increase in elapsed running time, while monitoring all accesses to all shared variables with reader sets of size two incurs a 140% to 350% increase in elapsed times.

The difference between the elapsed and CPU time increases stems from Amdahl's law [Amd67]. Almost all of the overhead of access anomalies detection is performed in parallel, and most of the benchmark programs contain a non-trivial amount of serial execution. The overhead of both maintaining concurrency information and monitoring accesses to shared variables could be greatly reduced by using static analysis to minimize the amount of monitoring actually performed. Even if this preprocessing is not performed, the increase in elapsed time is small enough to allow on-the-fly access anomaly detection to be a viable debugging tool.

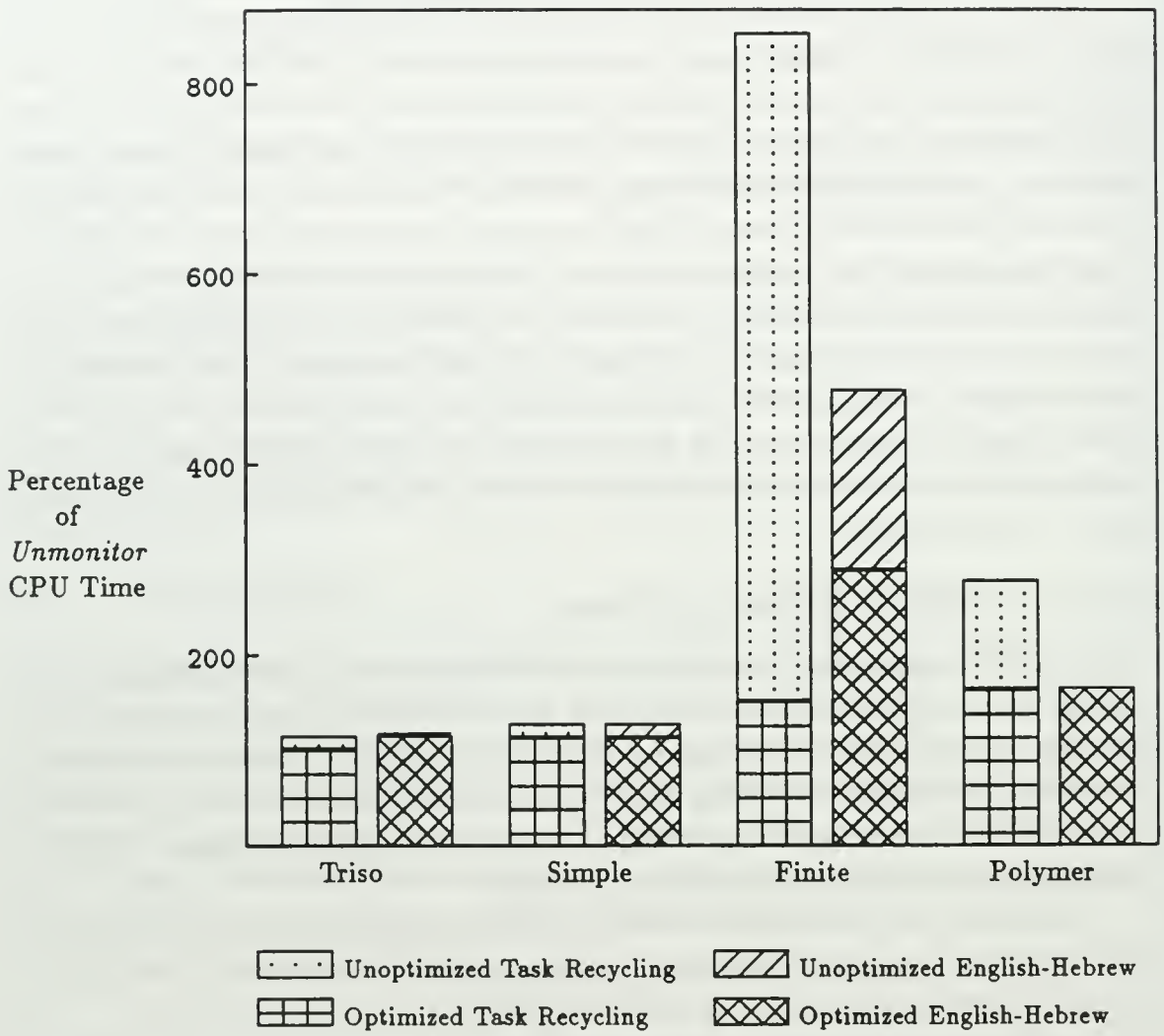


Figure 4.6: Percentage Increase in CPU Time for Maintaining Concurrency Information

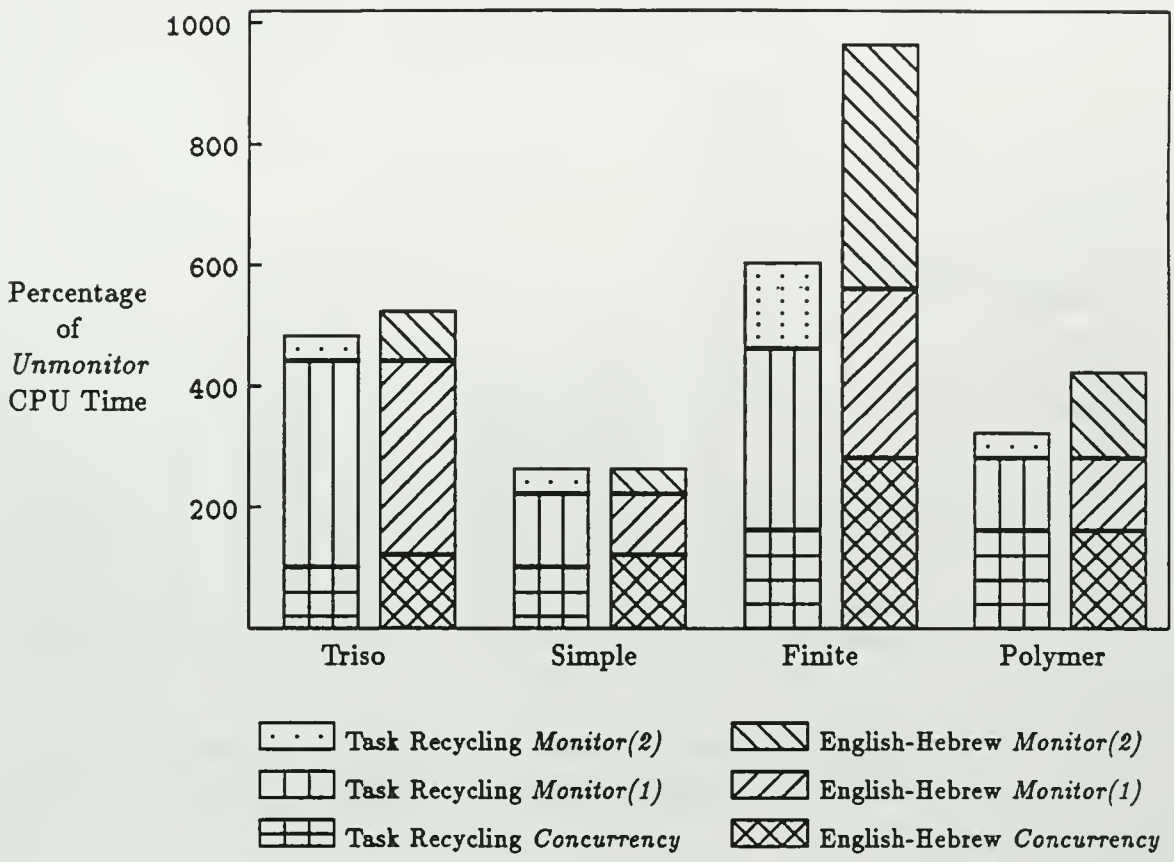


Figure 4.7: Percentage Increase in CPU Time for Monitoring Accesses

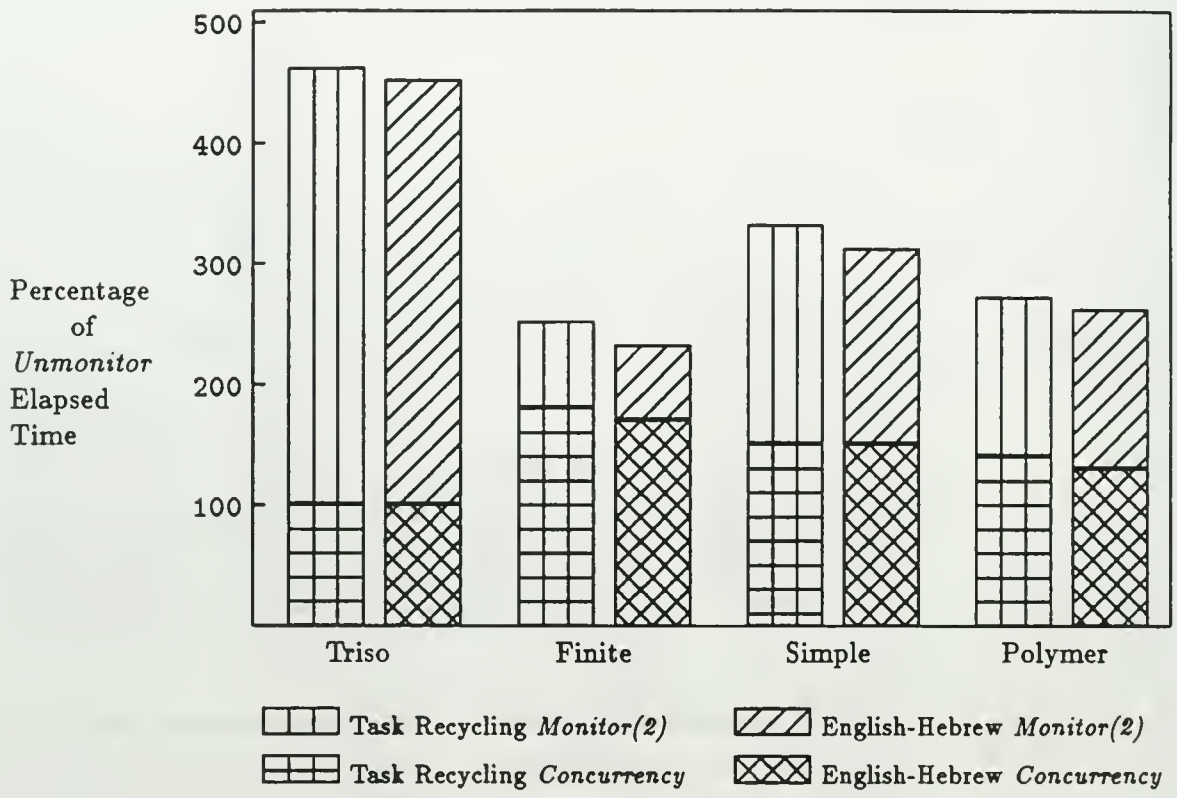


Figure 4.8: Percentage Increase in Elapsed Running Time

Chapter 5

Critical Section Coordination and Nondeterminism

The standard representation of critical section coordination, described in Section 2.3, adds coordination edges to the POEG to explicitly model the execution order of critical section instances. While this *ordered* representation is correct, anomaly detection is computationally expensive. Every execution order of the critical sections must be analyzed, since anomalies can be masked by the coordination edges used to model a given execution order.

In many cases, however, critical sections coordination is "well-behaved" in the sense that the execution order of critical sections does not affect the program execution. Consider

```
X := 0
doall i := 1 to n
    j := f(i)
    lock(L)
    X := X + j
    unlock(L)
endall
T := X
```

Figure 5.1: Order Independent Program

the program in Figure 5.1. The critical sections in this program compute the value $f(1) + f(2) + \ldots + f(n)$. This sum is independent of the order in which the terms are added. Since only the final result is used, the outcome of the program is unaffected by the order in which the critical sections execute. In this chapter we propose an alternative *unordered* representation for critical section coordination that does not model the critical section execution order in the POEG and describe criteria for determining when this new representation is reliable.

The efficiency and effectiveness of dynamic access anomaly detection is greatly improved when the unordered representation is used.

1. Less ordering is added to the POEG so that more anomalies are exposed in a given execution instance.

2. Theorem 5 applies when a program P contains no other unmatched coordination. Thus, a single execution instance is sufficient to prove that P is anomaly free for an input vector I. (If the ordered representation is used, $N!$ execution instances must be analyzed where N is the number of concurrent critical sections.)

In addition, critical sections that are reliably modeled by the unordered representation can be ignored in program analysis and debugging. For instance, these critical sections do not have to be traced by trace-and-replay debuggers, since their execution order does not affect subsequent execution. Section 5.1 describes the unordered representation of critical section coordination.

Unfortunately, the unordered representation does not always reliably model the interaction between the critical sections and the remainder of the program. Section 5.2 addresses the problem of determining when a program's execution is dependent on the execution order of critical sections. Specifically, the notion of nondeterminism stemming from the execution order of critical sections is refined and three independent types of nondeterminism identified: *parallel*, *reference* and *sequential nondeterminism*. The unordered representation is guaranteed to be reliable if a set of critical sections is parallel, sequential and reference deterministic.

Section 5.3 presents compiler-based static analysis algorithms for detecting these three types of nondeterminism. (Besides determining the reliability of unordered representation, isolating any type of nondeterminism is useful for understanding the logic of a parallel program.) Parallel, reference and sequential nondeterminism can be detected relatively accurately. Only information about variables accessed within critical sections is needed, and critical sections are generally small sections of code that access a largely disjoint set of variables. In contrast, statically determining that a set of concurrent threads contain no access anomalies is relatively difficult, since this requires obtaining accurate information for every variable that is read and written in a concurrent thread. Thus, it may be possible to prove at compile time that a given program is parallel, reference and sequential deterministic, but not to show that it is anomaly free.

Figure 5.2 presents a block diagram of a general nondeterminism analysis system. The output from the static analysis phase (comprised of parallel, sequential, and reference nondeterminism detection and static access anomaly detection) is used by the dynamic access

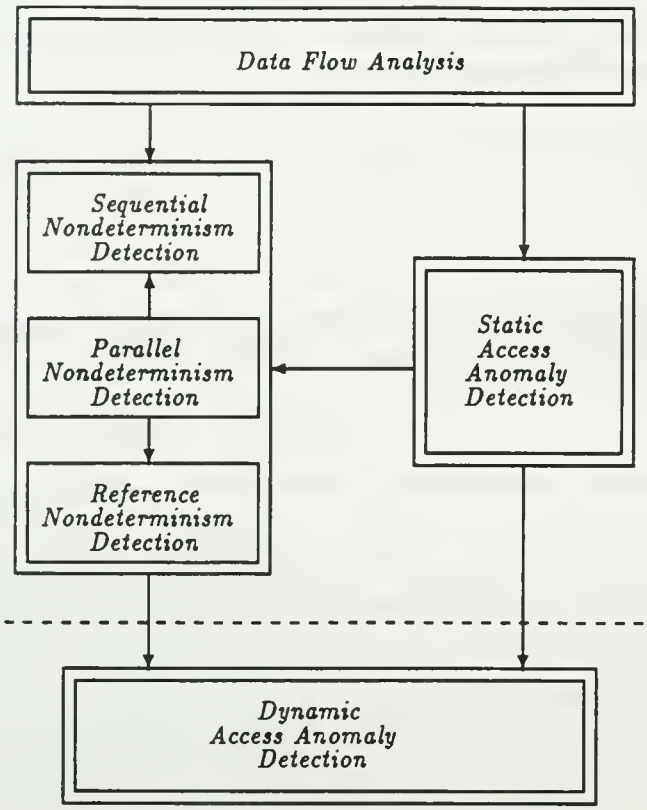


Figure 5.2: Nondeterminism Detection System

anomaly detection system to decide: (i) which representation to use for critical section coordination, and (ii) which accesses are potential anomalies and must be monitored.

Heuristics can make access anomaly detection tractable even in the presence of nondeterminism. For instance, a practical dynamic access anomaly detection system will generally not attempt to prove that a program and input vector pair is anomaly free, since this is often intractable. When the only correctness requirement of a dynamic anomaly detection system is that no false anomalies are reported, certain types of nondeterminism can be ignored. Section 5.4 presents several alternative representations of critical section coordination that make dynamic anomaly detection more effective through a better classification and semantic understanding of the critical section coordination in a program.

5.1 Unordered Critical Sections Representation

The ordered representation for critical sections, presented in Section 2.3, models the execution order of the critical sections explicitly by adding coordination edges to the

POEG. The coordination edges from unlock operations to lock operations represent an arbitrary ordering of events, not an ordering that is intended or necessary. Thus, the ordered representation does not accurately reflect the semantics of critical sections. Often, more ordering is added than intended by the programmer, making anomaly detection more expensive than necessary.

The *unordered* representation is an alternative way of modeling critical section coordination. The unordered representation ignores the execution order of critical sections so that no coordination edges are added to the POEG. The benefit of the unordered representation is that it compresses many POEGs into a single POEG, thereby exposing anomalies that otherwise can be detected only by examining many execution instances. Moreover, any critical section that can be reliably modeled by the unordered representation is matched. (Refer to Section 2.4 for the definition of matched.) Since Theorem 5 applies to this class of critical sections, a single execution instance is sufficient to prove that a program with unordered critical sections and no other nondeterminism is anomaly free for a given input vector.

Because no coordination edges are added to the POEG, an alternative method must be used to ensure that accesses made within concurrent critical sections are not incorrectly reported as access anomalies. It is insufficient simply to not monitor these accesses, since anomalies must be reliably reported when accesses inside and outside critical sections conflict. Specifically, read events performed within a critical section conflict with simple write events (e.g. write events performed outside a critical section), and a write performed within a critical section conflicts with simple read and write events. Accesses within critical sections can be correctly monitored through the use of *lock covers*[1].

By the semantics of critical sections, every access in a critical section is protected by the lock associated with the critical section. A *lock cover* is associated with each access and contains all of the locks that are held when the access is performed. An access is covered by more than one lock when critical sections are nested: an access has an empty lock cover if it is not performed in a critical section.

Lock covers are used in detecting conflicts between accesses. Accesses that are covered by the same lock are guaranteed not to execute at the same time–regardless of concurrency relationships of the blocks performing the accesses–because of the semantics of critical sections. Therefore, any two accesses that have some lock in common never conflict. Two accesses that do not have a lock in common are treated as simple read and write events. This representation is robust in that it allows for the detection of critical sections that were neglected to be locked or inadvertently protected by an incorrect lock.

[1]Lock covers are similar to the notion of *flavors* proposed by Callahan and Kennedy for the static analysis of critical sections [CK87].

Bernstein's conditions can be modified to reflect the semantics of lock covers. Let $Write_L$ denote the set of write events with lock cover L and $Write_{\overline{L}}$ denote the set of write events that are not covered by any lock in L. The set of simple write operations is $Write_L$ where $L = \emptyset$. (In this case, $\overline{L}$ contains all locks.) $Read_L$ and $Read_{\overline{L}}$ are defined similarly. The conditions for detecting access anomalies between two concurrent blocks S_i and S_j are as follows:

Modified Bernstein's Conditions:

$$Read_L(S_i) \cap Write_{\overline{L}}(S_j) = \emptyset$$
$$Write_L(S_i) \cap Read_{\overline{L}}(S_j) = \emptyset$$
$$Write_L(S_i) \cap Write_{\overline{L}}(S_j) = \emptyset$$

The execution order of S_i and S_j does not introduce any access anomalies if and only if the above three conditions are met for every lock cover L.

To illustrate the use of lock covers, consider the POEG in Figure 5.3 that models an execution instance of a modified version of the program fragment in Figure 2.8. (Accesses

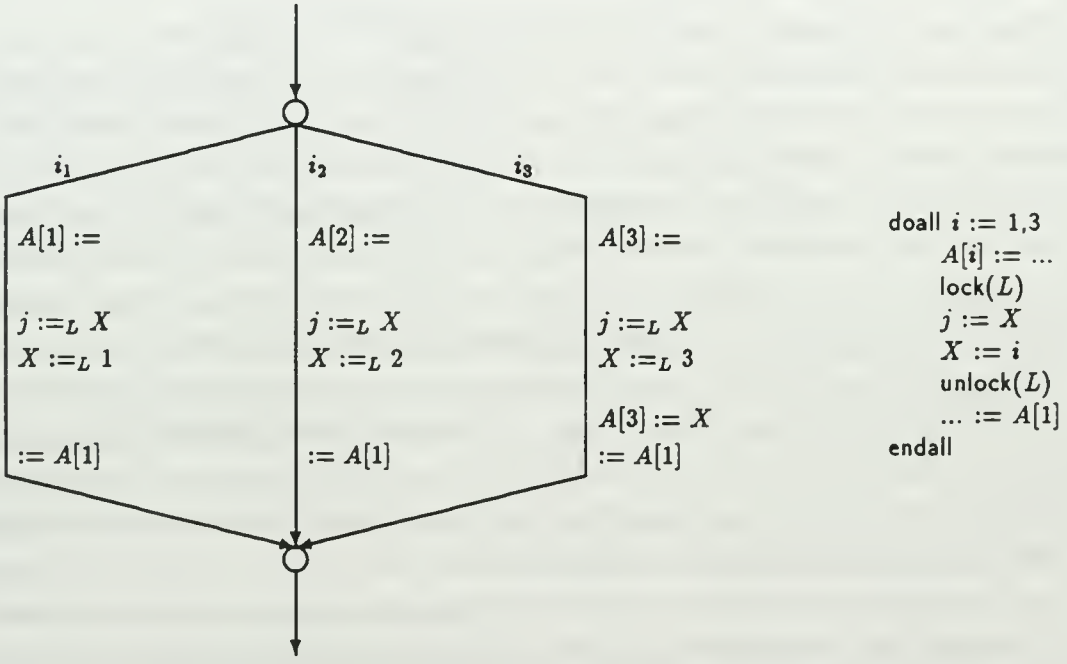


Figure 5.3: Unordered Representation of Critical Section Coordination

with lock cover $\{L\}$ are denoted by "$=_L$" in the POEG.) In this execution instance, iterates i_1, i_2 and i_3 enter the critical section in that order. The accesses of X inside the critical section are not reported as access anomalies even though they are concurrent, since all events have the same lock cover, $\{L\}$, and therefore do not conflict. However, the read of X outside the critical section by i_3 is correctly detected as an access anomaly

with respect to the critical section write operations. Moreover, the unsafe accesses to $A[1]$ are detected in *every* execution instance, regardless of the order in which iterates i_1, i_2 and i_3 enter the critical section.

5.1.1 Access Histories and Lock Covers

The algorithms for detecting access anomalies and updating the access histories must be extended to consider lock covers. Each entry in an access history contains information about the lock cover associated with the access. Since most events are not performed within critical sections, each access history has distinct *CS-reader* and *CS-writer* sets that record read and write events with non-empty lock covers. Distinguishing between simple and critical section read and write events minimizes the number of set operations performed when checking and updating an access history. It also saves space, since an empty lock cover is not stored with simple read and write events.

When a block executes a read event in a critical section, the *Check-CS-Read* algorithm, shown in Algorithm 5.1, checks for access anomalies. In particular, a critical section read event e conflicts with all events in the writer set that are concurrent with e and all events w in the CS-writer set that are concurrent with e such that there is no lock that covers both e and w. A simple read operation e conflicts with all events in the writer and CS-writer sets that are concurrent with e, as shown in algorithm *Check-Read*. A similar pair of algorithms, *Check-CS-Write* and *Check-Write*, are performed to check simple write and critical section write events.

Lock covers complicate the subtraction optimization for access histories. In particular, it is no longer the case that every future event that conflicts with an ancestor of the current event e also conflicts with e. To illustrate this, consider the POEG in Figure 5.4. Suppose that blocks b_1, b_2, b_3 and b_4 write X in that order and the access history for X is initially empty. After block b_1 writes X, an event with lock cover $\{A\}$ is added to CS-Writer(X). Suppose when block b_2 writes X, the event in CS-Writer(X) is overwritten with lock cover $\{A, B\}$. When block b_3 writes X, the conflict with the write of X in b_1 will not be detected. Alternatively, suppose after block b_2 writes X, the event in CS-Writer(X) is overwritten with the intersection of the lock covers of b_1 and b_2; namely $\{A\}$. When block b_4 writes X, it will be incorrectly detected as conflicting with the write by b_2.

Instead, an event a in the access history can be subtracted by the current access e when either: (1) a conflicts with e, or (2) a was performed by an ancestor of e and $\mathit{Locks}(e) \subset \mathit{Locks}(a)$. In both cases, an entry a is deleted only if any future access that conflicts with a also conflicts with the current access.

After a block performs a read event in a critical section, it updates the access history using the *Subtract-CS-Read* algorithm shown in Algorithm 5.2. It subtracts all ancestors

```
procedure Check-CS-Read(b, X)
    if IsConcurrent(b, Writer(X)) then report Access Anomaly
    for all a in CS-Writer(X) do
        if Locks(b) ∩ Locks(a) = ∅ and IsConcurrent(b, a) then
            report Access Anomaly
    endfor
end procedure

procedure Check-Read(b, X)
    for all a in CS-Writer(X) ∪ Writer(X) do
        if IsConcurrent(b, a) then report Access Anomaly
    endfor
end procedure

procedure Check-CS-Write(b, X)
    for all a in Reader(X) ∪ Writer(X) do
        if IsConcurrent(b, a) then report Access Anomaly
    endfor
    for all a in CS-Writer(X) ∪ CS-Reader(X) do
        if Locks(b) ∩ Locks(a) = ∅ and IsConcurrent(b, a) then
            report Access Anomaly
    endfor
end procedure

procedure Check-Write(b, X)
    for all a in Reader(X) ∪ CS-Writer(X) ∪ CS-Reader(X) ∪ Writer(X) do
        if IsConcurrent(b, a) then report Access Anomaly
    endfor
end procedure
```

Algorithm 5.1: Check for Read and Write Events with Lock Covers

from *CS-Reader(X)* that has some lock in common with its lock cover. Read events simply subtract all ancestors from the reader and CS-reader sets, as shown in *Subtract-Read*. Similar algorithms are performed for subtracting access histories for critical section write and simple write events.

A consequence of the unordered representation is that the size of access histories will tend to be larger than when the ordered representation is used. First, CS-reader sets are likely to contain more entries than reader sets since the programming paradigms that limit reader set sizes do not apply to accesses within critical sections. A variable is generally protected by a critical section because it is accessed by many concurrent threads. In the ordered representation, these accesses would be ordered and therefore subtracted from the access history.

Second, there can be nonconcurrent events in a CS-reader or CS-writer set. The size

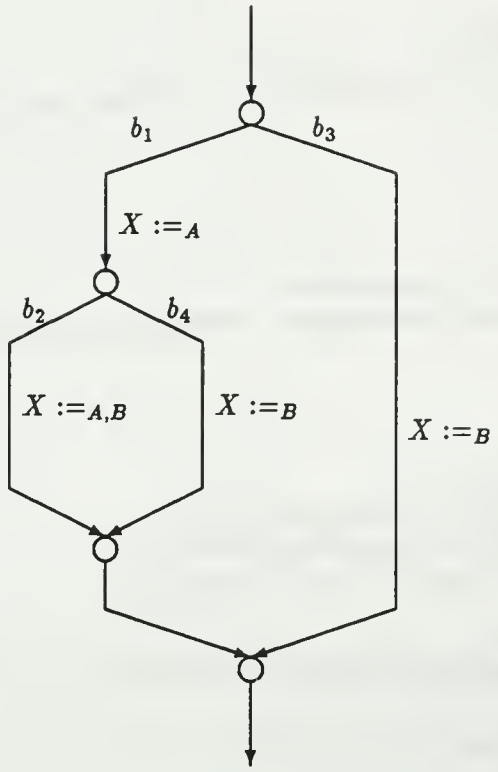


Figure 5.4: Subtraction Example

of the CS-reader and CS-writer sets are a function of the power set of the number of locks. (The upper bound on the number of events in an access history associated with any given lock cover is still bounded by the maximum concurrency of the POEG for CS-reader sets and one for CS-writer sets.) It is possible for programs to have many locks; consider a lock array that protects individual rows of a matrix. However, it is reasonable to expect that a small number of different lock covers are associated with any given variable, since complex locking patterns can lead to deadlocks and introduce excessive serialization.

5.2 Unordered vs. Ordered Representation

Although the unordered representation reliably models certain critical sections, it can fail to meet both Reliability Properties 1 and 2. When the unordered representation does not correctly model the ordering intended by the programmer, false anomalies can be reported. Consider the program in Figure 5.5 in which the critical sections pass access control to entries of array A. There are no access anomalies in this program. However, in those execution instances in which iterate k executes the critical section immediately before iterate m, the write of $A[k]$ by iterate k and the read of $A[k]$ by iterate m will be

```
procedure Subtract-CS-Read(b, X)
    for all a in CS-Reader(X) do
        if Locks(b) ⊂ Locks(a) and not IsConcurrent(b,a) then delete a from CS-Reader(X)
    endfor
    add b to CS-Reader(X)
end procedure

procedure Subtract-Read(b, X)
    for all a in Reader(X) ∪ CS-Reader(X) do
        if not IsConcurrent(b,a) then delete a from Reader(X) or CS-Reader(X)
    endfor
    add b to Reader(X)
end procedure

procedure Subtract-CS-Write(b, X)
    for all a in CS-Reader(X) ∪ CS-Writer do
        if Locks(b) ⊂ Locks(a) and not IsConcurrent(b,a) then
            delete a from CS-Reader(X) or CS-Writer(X)
    endfor
    for all a in Reader(X) do
        if not IsConcurrent(b,a) then delete a from Reader(X)
    endfor
    add b to CS-Writer(X)
end procedure

procedure Subtract-Write(b, X)
    for all a in Reader(X) ∪ CS-Reader(X) ∪ CS-Writer(X) do
        if not IsConcurrent(b,a) then
            delete a from Reader(X) or CS-Reader(X) or CS-Writer(X)
    endfor
    Writer(X) := b
end procedure
```

Algorithm 5.2: Subtraction for Read and Write Events with Lock Covers

falsely identified as an access anomaly if the unordered representation is used.

Similarly, anomalies can be missed if only a single execution instance is monitored. Consider the program in Figure 5.6. In those execution instances in which the first iterate to execute the critical section is not iterate 1, no anomaly occurs. However, in those execution instances in which iterate 1 is first, there is an access anomaly between write of $A[1]$ by iterate 1 in the critical section and the reads of $A[1]$ by iterates $2 \ldots n$ outside of the critical section.

Based on the preceding examples, one may (incorrectly) suspect that the unordered representation is reliable if and only if no variable is accessed inside and outside a critical section by concurrent threads. However, there are many cases where this simple test fails:

```
X := 0
doall i := 1 to n
    initialize(A[i])
    lock(L)
    j := X
    X := i
    unlock(L)
    process(A[j])
endall
```

Figure 5.5: Program with False Anomalies

```
init := false
doall i := 1 to n
    j := A[1]
    lock(L)
    if not init then A[i] := j
    init := true
    unlock(L)
endall
```

Figure 5.6: Program with Order Dependent Anomalies

1. The program in Figure 5.7 can be validly modeled by the unordered representation.

```
X := 0
doall i := 1 to n
    A[i] := i × X
    lock(L)
    X := X + 1
    unlock(L)
endall
```

Figure 5.7: Program with Order Independent Execution

   However, variable X is accessed inside and outside the critical section by concurrent threads.

2. The value of X in the program in Figure 5.8a is dependent on the execution order of the critical section. However, the only variable accessed inside and outside critical sections, X, is accessed by a block that is not concurrent with any critical section.

3. Similarly, the value of j in the program in Figure 5.8b is dependent on the execution order of the critical sections even though no variable is accessed both inside and outside the critical section.

The remainder of this section defines when the unordered representation is reliable.

```
X := 0
doall i := 1 to n
    lock(L)
    X := i - X
    unlock(L)
endall
T := X
```

(a)

```
doall i := 1 to n
    j := odd
    lock(L)
    X := X + i
    if X mod 2 = 0 then
        unlock(L)
        j := even
    else unlock(L)
endall
```

(b)

Figure 5.8: Two Programs with Order Dependent Execution

In most programs, execution within a critical section is determined in part by the critical sections that have already executed. However, the execution of the remainder of the program is affected only when this nondeterminism is propagated outside of the critical sections. We can distinguish between three ways the indeterminate behavior within critical sections can introduce nondeterminism in the rest of the program. The following definitions are based on an outermost parallel construct C and lock L, where C_L denotes the set of critical sections in C that are covered by lock L, and $C_{\overline{L}}$ denotes the set of blocks in C that are not covered by lock L[2] .

- **Parallel Nondeterminism**

  Definitions:

  A variable X in a critical section in C_L has an *indeterminate value* with respect to L if and only if the series of values assigned to X are based on the execution order of the critical sections in C_L.

  C is *parallel nondeterministic* with respect to lock L if and only if there is an indeterminate value V with respect to L and a block B in $C_{\overline{L}}$ such that:

  (i) B is control dependent on V, or

  (ii) V is used by B, and B is ordered with V in at least one execution instance[3].

More intuitively, parallel nondeterminism occurs when the order in which concurrent threads execute the critical sections in a parallel construct C affects the execution *within* C itself. For example, the program in Figures 5.5 is parallel nondeterministic. The critical sections are used to pass access control over the entries of array A among

[2]Only programs with parallel constructs are considered in this chapter: programs that have unconstrained fork and join operations require slightly different definitions, theorems and proofs.

[3]If a value V and B are always concurrent, then they will be detected as an access anomaly in every execution instance in which they both appear. Since our goal is to prove the nonexistence of access anomalies, ignoring the affect of V on B does not introduce any circularity.

the concurrent iterates. Thus, which entry of array A a thread T processes depends on which thread entered the critical section immediately before T.

All parallel constructs that contain one type of potential access anomaly–namely, a critical section write event and a simple read event in a block that follows another critical section–are parallel nondeterministic. However, anomalies with critical section read events, critical section write events that conflict with read events which do not follow another critical section, or critical section write events that conflict with other write events will not cause the parallel construct to be identified as parallel nondeterministic.

One may be tempted to use a stricter definition of parallel nondeterminism by modifying condition (ii) as follows:

> (ii′) V is used by B, and B is ordered with V in *all* execution instance.

Unfortunately, when this definition is used , the interaction among the parallel iterates is not correctly captured, and Lemma 10–and therefore Theorem 17 (see below)–does not hold.

- **Reference Nondeterminism**

  Definitions:

  A variable X in a critical section in C_L is an *indeterminate variable* with respect to L if and only if whether X is accessed is based on the execution order of the critical sections in C_L.

  C is *reference nondeterministic* with respect to lock L if and only if an indeterminate variable X with respect to L is accessed by a block B in $C_{\overline{L}}$, and B and an access to X are concurrent in at least one execution instance.

A program is reference nondeterministic when the order in which threads execute the critical sections in C affects whether or not an access A in a critical section occurs, and A is a potential access anomaly. Reference nondeterminism is illustrated by the program in Figure 5.6. The entry of array A which is written in the critical section depends on which iterate executes the critical section first. Conflicting access to $A[1]$ occurs only when iterate 1 executes the critical section first.

- **Sequential Nondeterminism**

  Definitions:

  A variable X in a critical section in C_L has an *indeterminate result* with respect to L if and only if the final value of X is based on the execution order of the critical sections in C_L.

  C is *sequential nondeterministic* with respect to lock L if and only if there is a block B that is a descendant of C and:

(i) an indeterminate value with respect to L is propagated to B, or
(ii) an indeterminate result with respect to L is used by B.

Sequential nondeterminism occurs when the execution order of the critical sections in C affects execution *after* C terminates. Sequential nondeterminism is illustrated by the program fragment in Figure 5.8a. In this program, variable X is computed in the critical sections and is used only after the parallel construct completes execution. The value of X depends on the critical section execution order, since subtraction is not associative. If the subtraction operations were changed to addition operations, this program would be sequential deterministic, since the final result of X would be deterministic.

These three types of nondeterminism are orthogonal: the conditions under which each occurs, the importance of their detection, and their detection algorithms are independent.

The concepts of parallel, reference and sequential nondeterminism are used to define when the unordered representation is sufficient to model a set of concurrent critical sections.

Theorem 17: If a parallel construct C is parallel, reference and sequential deterministic with respect to a lock L, then the unordered representation meets Reliability Properties 1 and 2.

Theorem 18: If a parallel construct C is parallel and reference deterministic with respect to a lock L, then the unordered representation meets Reliability Properties 1 and 2 within C and Reliability Property 1 outside C.

A direct consequence of Theorem 17 is that one execution instance is sufficient to guarantee that a POEG deterministic program and input vector pair (P, I) is anomaly free if P contains critical section coordination and is parallel, sequential and reference deterministic with respect to all locks. More generally, whenever a set of critical sections is parallel, reference and sequential deterministic, its execution order can be ignored in any program analysis or debugging.

Theorem 18 states that the anomaly detection *within* a parallel construct C is independent of subsequent sequential nondeterminism. If the unordered representation is used for a parallel construct C that is parallel and reference deterministic but sequential nondeterministic, no false anomalies will be reported anywhere in the program. However, anomalies may be hidden in a given execution instance.

Lemma 10 *In the absence of access anomalies detected when the ordered representation is used, if a parallel construct C is parallel deterministic with respect to L, then the execution of blocks in $C_{\overline{L}}$ are independent of the execution order of the critical sections in C_L.*

Proof. Suppose, for the sake of contradiction, this were not true for an execution instance E and the execution of a block in $C_{\overline{L}}$ is dependent on the execution order of the critical sections in C_L. Information about the execution order of the critical sections in C_L must be conveyed to some block in $C_{\overline{L}}$. In the absence of access anomalies, the only accesses within critical sections in C_L that are dependent on the execution order of the critical section are indeterminate variables, values and results (by definition).

Information can be conveyed during the execution of a program only by modifying the control flow or data flow of the program. Both of these cases are addressed below:

- Since C is parallel deterministic with respect to L, no block in $C_{\overline{L}}$ is control dependent on an indeterminate value in a critical section in C_L. Therefore, no information is passed by control flow.

- Since C is parallel deterministic with respect to L, any indeterminate value D with respect to lock L that reaches a use U in $C_{\overline{L}}$ is concurrent with U in all execution instances. If D reaches U in E, an access anomaly would have been detected; otherwise, no information about the execution order is conveyed (D could occur after U is used). Therefore, no information is passed by data flow.

In the absence of access anomalies, no information about the execution order of the critical sections C_L is conveyed to a block in $C_{\overline{L}}$ through control or data flow. Hence the execution of every block in $C_{\overline{L}}$ must be independent of the execution order. □

Lemma 11 *With all other sources of nondeterminism fixed, if a parallel construct C is parallel and reference deterministic with respect to lock L, and an access anomaly is detected in at least one execution instance when the ordered representation is used, then an anomaly is guaranteed to be detected in every execution instance when the unordered representation is used.*

Proof. If an anomaly occurs during an execution instance E when the unordered representation is used, the proof is complete. Otherwise, suppose for the sake of contradiction, that an anomaly is not detected in E when the unordered representation is used, but is detected in another execution instance E'.

Since C is parallel deterministic for L, by Lemma 10 the actions of each block in $C_{\overline{L}}$ are independent of the execution order of the critical sections in C_L. In particular, when all other sources of nondeterminism are fixed, the variables read and the variables written by every block in $C_{\overline{L}}$ are the same in E and E' upto and including the first access anomaly. Since C is reference deterministic, every variable in a critical section in C_L that

is a potentially access anomaly and is read (resp. written) in E is also read (resp. written) in E'. Thus, each block in C accesses the same set of variables in E and E'.

Since the unordered representation does not modify the POEG, and all other sources of nondeterminism are fixed, the ancestor relationship is identical for E and E'. Therefore, if an anomaly is detected in E', an anomaly must also be detected in E. □

Lemma 12 *If a parallel construct C is parallel and reference deterministic with respect to L, then anomalies will not be falsely reported when the unordered representation is used.*

Proof. Suppose, for the sake of contradiction, that an anomaly is falsely reported between blocks b_i and b_j in an execution instance E in which the critical sections in C_L are modeled by the unordered representation. Assume b_i happened to execute before b_j in E. (A symmetric argument holds if b_i executed after b_j.) Since there is no anomaly detected when the ordered representation is used, block b_i (or one of its descendants) is a critical section cs_i in C_L and b_j (or one of its ancestors) is a critical section cs_j in C_L such that cs_i executed immediately before cs_j in E.

Let E' be an execution instance with all sources of nondeterminism fixed in the same way as E except that cs_i executes immediately after cs_j in E'. This execution order must be able to occur since cs_j and cs_i are concurrent in the unordered representation of E. There are three different relationships between cs_i, cs_j, b_i and b_j in E' when the ordered

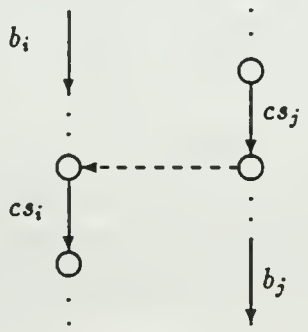


Figure 5.9: Partial POEG for E'

representation is used (Figure 5.9 shows a partial POEG for the ordered representation of E'):

1. If $cs_i \neq b_i$ then b_i and b_j are concurrent, since cs_j and all of its descendants are concurrent with all of the ancestors of cs_i.

2. If $cs_i = b_i$ and $cs_j \neq b_j$ then b_i and b_j are concurrent, since cs_i is concurrent with all of the descendants of cs_j.

3. If $cs_i = b_i$ and $cs_j = b_j$ then no anomaly could have been reported, since two accesses in critical sections covered by the same lock never conflict.

In all cases b_i and b_j are concurrent in the ordered representation of E'. By the same logic as the proof of Lemma 11, b_i and b_j access the same set of variables in E and E'. Therefore, an anomaly between b_i and b_j will also be detected in E' and hence is not a false anomaly. □

Lemma 13 *In the absence of access anomalies, if a parallel construct C is sequential deterministic with respect to L, then the program state after the execution of C is independent of the execution order of the critical sections covered by lock L in C.*

Proof. Suppose, for the sake of contradiction, that the program state after the execution of C were dependent on the execution order of critical sections in C_L and neither condition (i) nor condition (ii) in the definition of sequential nondeterminism was met. Because C is a parallel construct, the code that is executed on the exit of C cannot be control dependent on a condition in C.

There are two other cases to consider:

1. If C is parallel deterministic, then by Lemma 10 there are no values computed in the blocks in $C_{\overline{L}}$ that are affected by the execution order of the critical sections in C_L. Therefore, if a result value in a critical section in C_L is not propagated outside of the construct (i.e. condition (ii) is not met), the execution of subsequent code cannot be dependent on the execution order of critical sections in C_L.

2. If C is parallel nondeterministic, by Lemma 10 the only additional values computed in C that are affected by the execution order of the critical sections in C_L are those that are propagated from a use that meets condition (i). Therefore, if neither condition (i) nor (ii) is met, the code following C cannot be dependent on the execution order of critical sections in C_L.

Hence, the program state after the execution of the construct must be independent of the execution order. □

Theorem 17 *If a parallel construct C is parallel, reference and sequential deterministic with respect to a lock L, then the unordered representation meets Reliability Properties 1 and 2.*

Proof. Lemma 11 and Lemma 12 guarantee that the unordered representation will reliably detect access anomalies within C if C is parallel and reference deterministic with respect to L. Lemma 13 proves that the remainder of the execution is independent of

the execution order of the critical sections in C_L if C is sequential deterministic with respect to L. Therefore, the unordered representation correctly captures the coordination stemming from the critical sections covered by L in C. □

Theorem 18 *If a parallel construct C is parallel and reference deterministic with respect to a lock L, then the unordered representation meets Reliability Properties 1 and 2 within C and Reliability Property 1 outside C.*

Proof. Reliability within C follows directly from Lemma 11 and Lemma 12. In addition, no false anomalies can be reported in code following C since it is not concurrent with any access in C. □

5.3 Detecting Parallel, Reference and Sequential Nondeterminism

This section presents algorithms that use compiler-based static analysis techniques to detect sequential, parallel and reference nondeterminism. As in most static analysis, there is a trade-off between efficiency and an exact answer. Several assumptions are made that greatly reduce the amount of work performed at the expense of increased inaccuracy. As a consequence, every nondeterministic parallel construct will be detected. However, a parallel construct that is sequential deterministic, for example, may be identified as sequential nondeterministic.

The first step in detecting parallel, reference and sequential nondeterminism is to determine how computation within critical sections is propagated to the remainder of the program. The portion of the code that is reached during this propagation step is the only part of the program that is potentially affected by execution within the critical sections. Once this information is obtained, we check individually for parallel, sequential, and reference nondeterminism. The general outline for these algorithms is given below.

Suppose we want to test a parallel construct for a given type of nondeterminism, for instance, sequential nondeterminism. We first check if a necessary reaching condition is met:

> **Reaching Condition.** Values computed within a critical section are propagated outside of the critical section.

If the reaching condition is not met, the parallel construct is sequential deterministic, since there is no interaction between the critical sections and the remainder of the program.

Otherwise, each access that meets the reaching condition is further analyzed to see if a necessary order dependency condition is met:

Order Dependency Condition. Values propagated outside critical sections are dependent on the order in which critical sections are executed.

Evaluating order dependency determines whether or not the propagated value is dependent on the execution order of the critical sections. The parallel construct is guaranteed to be sequential deterministic if either the reaching or order dependent condition is not met.

Before describing the algorithms, we briefly define the data structures used by the static analysis algorithms: the parallel control flow graph and program dependence graph. To be compatible with current compiler optimization literature, read events are referred to as *uses* and write events are referred to as *definitions*. In the remainder of this section, the terms block, critical section and parallel, reference and sequential nondeterminism are defined with respect to a given lock L. However, for ease of exposition, the phrases " not covered by lock L", "covered by lock L" and "with respect to lock L" will not be stated explicitly.

Parallel Control Flow Graph

A *parallel control flow graph* represents an explicitly parallel program much in the same way that a sequential control flow graph represents a sequential program. Nodes in the parallel control flow graph correspond to sequences of instructions with one entry point, one exit point and no internal fork, join and coordination operations. Examples of parallel control flow graphs include the synchronized control flow graph defined by Callahan and Subhlok [CS88] and the annotated flow graph of Taylor [Tay83,TO80]. The following algorithms use two types of information obtained from a parallel control flow graph: nodes that can potentially execute concurrently, and the set of definitions that reach a given use.

It is difficult to compute the concurrency relationship exactly. One way to approximate the concurrency relationship is by computing the SCPreserved set of Callahan and Subhlok [CS88,CKS90]. A node n is in SCPreserved(x) only if n precedes x in every execution instance in which they both appear. Using SCPreserved sets, the concurrency relationship is computed as follows:

$$\text{Concur}(n) = \{\ x : x \notin \text{SCPreserved}(n) \wedge n \notin \text{SCPreserved}(x)\}$$

This equation is conservative: a node x is in Concur(n) if x and n are concurrent. However, a node y which is never concurrent with n might also be in Concur(n).

In parallel programs there are two types of definitions that can reach a given use:

Reach$_S$(u) : the definitions that reach use u from nodes that are ordered with u in at least one execution instance.

Reach$_C$*(u)* : the definitions that reach use u from nodes concurrent with u in at least one execution instance.

(Note that a definition can be in both the *Reach*$_C$ and *Reach*$_S$ sets of a given use.) Only the definitions in the *Reach*$_S$ set are required in detecting parallel, sequential and reference nondeterminism.

Reach$_S$ sets are calculated from the parallel control flow graph in the same manner as in sequential data flow analysis (described in [AU77]). However, an extended notion of predecessor is used to correctly model critical section coordination. In any execution instance, a value that reaches the end of a critical section instance can be safely used by the subsequent critical section instance. Therefore, a node p that is concurrent with a node n is treated as an additional predecessor of n if three conditions hold: (i) p and n are covered by the same lock L, (ii) p has successors not covered by L, and (iii) n has predecessors not covered by L.

Program Dependence Graph

The second data structure used is the *program dependence graph* proposed by Ferrante, Ottenstein and Warren [FOW87]. Program dependence graphs allow for simpler representation of certain programming constructs and have proved invaluable in many different uses in static analysis of programs ranging from from parallelization [ABC+88],[BCF+88],[BHRB89] and program analysis [Sel89] to creating program slices [HPR87],[HRB88]. Every node in a program dependence graph corresponds to a node in the parallel control flow graph. There are *data dependence* and *control dependence* edges between nodes.

There is a *control dependence* edge from node n to node c with label T if c (eventually) executes whenever the predicate at the end of n has value T. An array reference $A[i]$ is represented in the control dependence graph as being control dependent on the index variable i. The initial basic blocks of a concurrent thread has the same control dependence relationship as its associated fork operation. Every program dependence graph has an embedded control dependence graph, which consists of control dependence edges, and a forward control dependence graph [CHH89] which is the control dependence graph with all back edges removed.

In contrast to control flow graphs, data dependences are explicitly represented in a program dependence graph. There is a *data dependence edge* in a program dependence graph to a use u from all definitions in *Reach*$_S(u)$. (A definition that reaches a use u but is always concurrent with u is not represented by a data dependence edge in the program dependence graph.) Several algorithms exist for building a program dependence graph [FOW87,CFR+89].

To illustrate the structure of a program dependence graph, consider the program

fragment and associated program dependence graph in Figure 5.10. Data dependences are denoted in the program dependence graph by dashed lines and control dependences by solid lines; the control dependence labels are shown in bold face. In this program

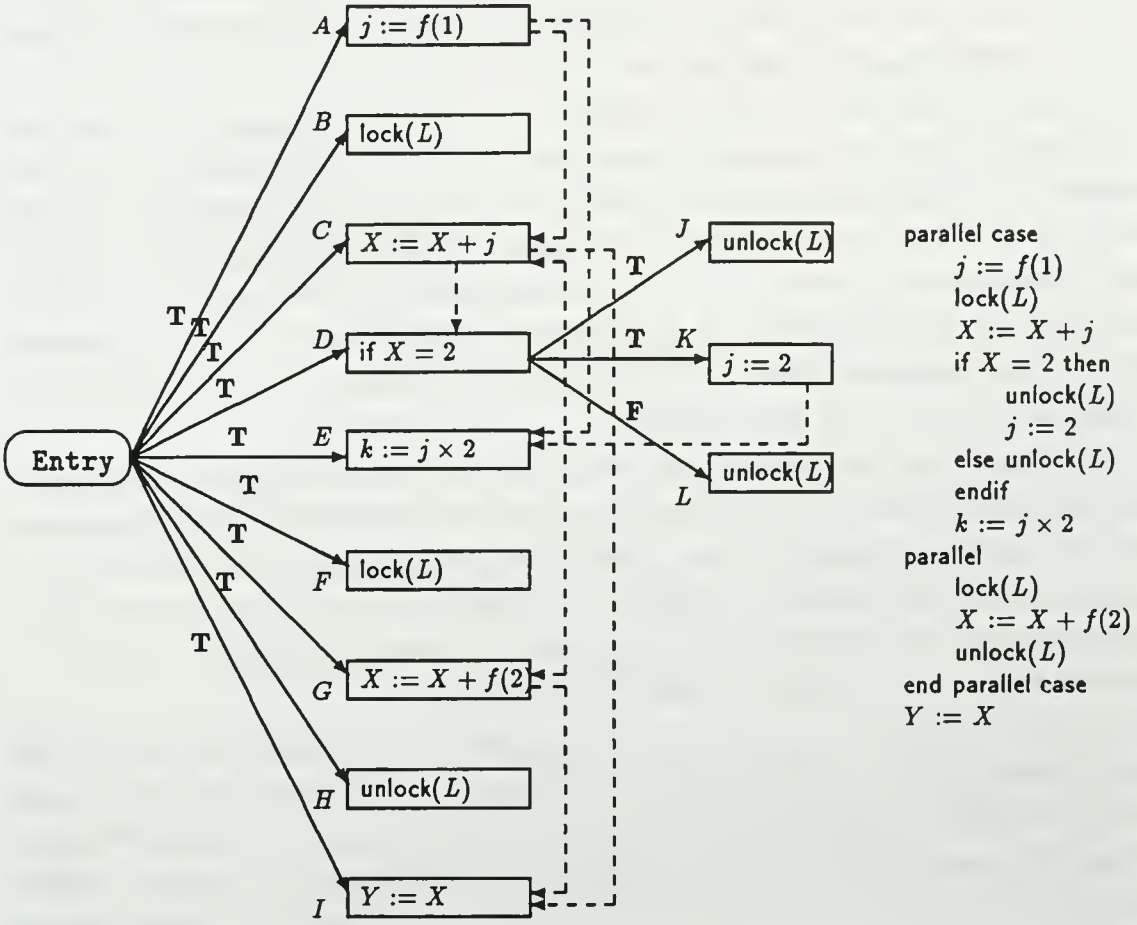


Figure 5.10: Program Dependence Graph

dependence graph, every node is a single statement. Node K is control dependent on D because it executes whenever the predicate at the end of node D has value T. Node C is data dependent on A because the use of j in C is reached by the definition of j in A.

5.3.1 Propagating Critical Section Indeterminacy

Our first goal is to identify all potentially indeterminate behavior within critical sections, and see how this affects the remainder of the program. In doing so we make the first simplifying assumption:

Assumption 1. Every definition in a critical section has an indeterminate value.

It is possible for a value V computed within a critical section to be independent of the critical section execution order. However, since the amount of sequentialization incurred by a critical section is linear in the size of the code, actions performed within the critical section are generally directly dependent on the current state of the variables protected by the critical section. Otherwise, the computation of V could be moved outside the critical section, thereby improving the code.

We propagate information about critical section indeterminate computation using program slices. A *slice* of a program is defined with respect to some program point P and variables X and consists of all of the definitions and predicates of the program that might affect the value of X at point P [Wei84]. Similarly, a *forward slice* of a program is defined with respect to some program point P, and consists of all points and variables that might be affected by the computation at P. Thus, $a \in slice(b)$ if and only if $b \in forward\text{-}slice(a)$. By computing forward slices we identify all code that is possibly affected by computation at some step in the program.

We create a forward slice from each definition with the critical section by traversing the program dependence graph, following the data dependence and control dependence edges. If a node n is reached on the traversal, then every predicate and definition within n is part of the forward slice. Therefore, a program dependence graph in which each node is as small as possible–for example, a single statement rather than a sequence of statements–allows more accurate results to be computed.

During the traversal, every node in the program dependence graph has one of three states associated with it: *deterministic*, *indeterminate* or *nondeterministic*. Initially, all nodes in the program dependence graph that are in critical sections and contain definitions are marked indeterminate (based on Assumption 1). If a critical section node n is reached during the traversal by only other critical section nodes, the state of n is indeterminate. Otherwise, if a node n is reached during the traversal, its state is nondeterministic. The distinction between indeterminate and nondeterministic nodes is important when detecting sequential and reference nondeterminism. For instance, if the state of the root node of a critical section A is nondeterministic, then whether or not A executes at all in a given execution instance can depend on the execution order of preceding critical sections.

Routine *Forward-Slice* in Algorithm 5.3 is used to propagate potential indeterminacy throughout the program. The amount of work required by routine *Forward-Slice* is the cost of traversing the program dependence graph three times, $O(|PDG_C|)$. It follows directly from results by Ottenstein and Ottenstein [OO84] that every potentially indeterminate node will be detected by creating forward slices.

To illustrate the propagation of indeterminacy, consider the program dependence graph in Figure 5.10. Before the traversal, nodes C and G are marked indeterminate

```
procedure Forward-Slice(C, L)
    for all n ∈ PDG_C do
        if L ∈ Locks(n) and Def(n) ≠ ∅ then
            state(n) := indeterminate
        else state(n) := deterministic
    endfor
    for all n ∈ PDG_C such that L ∈ Locks(n) and Def(n) ≠ ∅ do
        Traverse(n)
    endfor
end procedure

procedure Traverse(n)
    for all children c of n do
        if state(c) < state(n) then
            if L ∉ Lock(c) then
                state(c) := nondeterministic
            else state(c) := state(n)
            Traverse(c)
        endif
    endfor
end procedure

        { Note: deterministic < indeterminate < nondeterministic}
```

Algorithm 5.3: Propagating Potential Indeterminacy

since they are critical sections nodes and they contain definitions. After the traversal, the set of indeterminate nodes is $\{C, D, G, J, L\}$; the set of nondeterministic nodes is $\{K, E, I\}$. Node K is nondeterministic since it is control dependent on D, node E is nondeterministic since it is data dependent on K, and node I is nondeterministic since it is data dependent on nodes G and C.

5.3.2 Detecting Parallel Nondeterminism

We can re-formulate the definition of parallel nondeterminism in terms of reaching and order dependent conditions :

Reaching Condition: There is a use U in a block in C, a definition or predicate D in a critical section in C, and $D \in Reach_S(U)$ or U is control dependent on D.

Order Dependency Condition: D has an indeterminate value.

Clearly, if there are no uses in a parallel construct C that are reached by critical section definitions or dependent on critical section predicates, C is parallel deterministic.

However, even when the reaching condition is met, the propagation of critical section indeterminate behavior makes the detection of parallel nondeterminism very simple.

Because of Assumption 1, every indeterminate definition within C is assumed to be an indeterminate value. Thus, the algorithm for detecting parallel nondeterminism is trivial. Every access A that is in a nondeterministic node in C is parallel nondeterministic. The parallel construct C is parallel nondeterministic if there is a parallel nondeterministic node that is not in a critical section.

To illustrate the detection of parallel determinism, consider the parallel nondeterministic program and associated program dependence graph in Figure 5.10. Nodes K and E are parallel nondeterministic, since they are nondeterministic and in the parallel construct C. Moreover, because they are not part of a critical section, the parallel construct is identified as parallel nondeterminism. (Node I is not parallel nondeterministic since it is not in C.)

5.3.3 Detecting Sequential Nondeterminism

Similar to parallel nondeterminism, we can formulate the definition of sequential nondeterminism in terms of reaching and order dependent conditions as follows:

> **Reaching Condition:** There is a use U that is a descendant of C, a definition $D \in Reach_S(U)$ and D is in C.
>
> **Order Dependency Condition:** Either (1) D in a critical section and is an indeterminate result, or (2) D is parallel nondeterministic.

Clearly, if there is no use outside of the parallel construct C that is reached by a definition within C, the parallel construct is sequential deterministic. Moreover, any use that meets order dependence condition 2 is pessimistically assumed to be sequentially nondeterministic (based on Assumption 1). If neither case holds, further analysis is required to determine if order dependent condition 1 is met.

In contrast to indeterminate values, definitions within critical sections are often determinate results. Consider for example, a program for computing a maximum in parallel. Although the value of the maximum at any intermediate step is indeterminate, the final result is always the same.

Unfortunately, determining whether or not a definition is an indeterminate result can be quite expensive. All possible critical section execution orders and all paths through each critical section must be analyzed. (Since D is not parallel nondeterministic, all variables that are used to compute D have the same initial value in any execution instance and the same set of critical sections are guaranteed to execute in all execution instances.) If there are N concurrent critical sections with P paths per critical sections, $(PN)!$ execution orderings must be analyzed. This is clearly computationally intractable.

Rather than examine all execution orders, the following simplifying assumption is made:

Assumption 2. A definition D is an indeterminate result if the value of D is dependent on the execution order of at least one pair of critical section instances.

D is a determinate result if the value of D after the execution of every pair of critical sections is independent of their execution order. When this property holds, the execution order of any two adjacent critical sections can be exchanged in any sequentialization of the critical sections without changing the overall result of the execution. Therefore, an arbitrary execution order can be produced by performing a series of exchanges without changing the final result.

Assumption 2 reduces the number of critical section execution orders that must be analyzed to $O(PN^2)$. This overhead is further mitigated by two factors. First, critical sections are generally quite small. Thus, there is little code to generate and very few paths to consider. Second, in homogeneous parallelism–such as that stemming from a doall operation–the code for all critical sections is identical for all parallel threads. Therefore, a pair of generic instances can be analyzed to determine if the order dependence criterion is met.

To evaluate order dependence, each pair of critical sections A_i and A_j is analyzed using symbolic execution [CR84]. Two sequential instruction sequences are created for each pair: one in which A_i is followed by A_j and another in which A_j is followed by A_i. For each path through the critical sections, the two instruction sequences are symbolically executed. For each path, the pair of final values of definition D are converted to a canonical form and are verified to be computationally identical. In general, a program is sequential deterministic only if the operations performed in the critical section are associative and commutative.

To illustrate the detection of sequential nondeterminism, consider the program shown in Figure 5.10. In this program the critical sections compute the sum $f(1) + f(2)$. Only node I meets the reaching condition. Since node I does not meet order dependent condition 2, order dependence condition 1 must be evaluated.

To do so the following two instruction sequences are created:

$A_1 \parallel A_2$:	$A_2 \parallel A_1$:
$X := X + j_1$	$X := X + f(2)$
$X := X + f(2)$	$X := X + j_1$

Using symbolic execution, the final values of the variable X for these two instruction sequences are computed to be the following:

$$X := (X + j_1) + f(2) \qquad X := (X + f(2)) + j_1$$

These two expressions are computationally equivalent. Therefore, the above program is sequential deterministic.

5.3.4 Detecting Reference Nondeterminism

Reference nondeterminism is the most complicated type of nondeterminism to detect. We can formulate the definition of reference nondeterminism in terms of reaching and order dependent conditions as follows:

> **Reaching Condition:** There is an access A_B of variable X in a block in C, an access A_C of variable X in a critical section in C, and A_B or A_C is a definition.
>
> **Order Dependency Condition:** X is an indeterminate variable.

Clearly, C is reference deterministic if either: (i) there are no potential anomalies with critical section accesses, or (ii) there is no conditional execution within critical sections. Otherwise, a more sophisticated analysis is required.

To evaluate the reaching condition, the references in critical sections that are potential access anomalies are isolated. In particular, the following two sets are computed for each critical section A:

> $Pot_R(A)$: the variables read in A that are potential anomalies
> $Pot_W(A)$: the variables written in A that are potential anomalies

Static analysis can be used to detect potential anomalies [CS88], or $Pot_R(A)$ and $Pot_W(A)$ can be conservatively assumed to contain every variable accessed in A.

To evaluate order dependence, each critical section is analyzed to determine if the same set of potentially anomalous variables are read and written regardless of the critical section execution order. To do so, the following sets are computed for each critical section node n in A:

> $May_R(n)$: the variables in $Pot_R(A)$ with read events control dependent on n
> $Must_R(n)$: the variables in $Pot_R(A)$ read in every execution that includes n

The algorithm for computing May_R and $Must_R$ sets–shown in Algorithm 5.4–uses a forward control dependence graph. (The definitions and algorithms for May_W and Must_W sets are similar to those for computing May_R and Must_R but use the *Definition* set.) It is shown in [CHH89] that a forward control dependence graph for a reducible program is a dag. Therefore, the *May* and *Must* sets can be computed by performing forward and backward topological traversals of the forward control dependence graph.

Reference nondeterminism is detected by verifying that every variable in $May_R(n)$ is also in $Must_R(n)$ for all nodes n in A that are not deterministic. In other words, we check

```
procedure Compute-May_R(A)
    for all node n in a backward topological traversal of the FCDG_A do
        May_R(n) := Use(n) ∩ Pot_R(A)
        for all children c of n do
            May_R(n) := May_R(n) ∪ May_R(c)
            if c is a True child of n then
                True(n) := True(n) ∪ Both(c)
            else False(n) := False(n) ∪ Both(c)
        endall
        Both(n) := (True(n) ∩ False(n)) ∪ Def(n)
    end for
end procedure

procedure Compute-Must_R(A)
    for all nodes n in a forward topological traversal of the FCDG_A do
        Must_R(n) := Use(n) ∩ Pot_R(A)
        for all parents p of n do
            if n is a True child of p then
                Must_R(n) := Both(n) ∪ Must_R(p) ∪ True(p)
            else Must_R(n) := Both(n) ∪ Must_R(p) ∪ False(p)
        endall
    endall
end procedure
```

Algorithm 5.4: Computing May_R and $Must_R$

to see if there is a potential access anomaly that occurs for some value of the predicate of n but is not guaranteed to occur in every execution that includes n. The May_W and $Must_W$ sets are defined and used symmetrically.

There are two special cases that must be considered: (1) the root block of critical section A is parallel nondeterministic, and (2) critical section A has internal coordination nodes. In case 1, critical section A may never be executed at all; in case 2, the two accesses may have different concurrency relationships. Both cases are handled by conservatively setting $Must_R(n)$ to be empty for all n in A. The routine in Algorithm 5.5 is performed for each critical section A to check for reference nondeterminism.

The complexity of detecting reference nondeterminism is five traversals of the control dependence graph times the number of shared variables. Theorem 19 proves that *Check-Reference* will not miss any reference nondeterministic constructs.

To illustrate the detection of reference nondeterminism, consider the control dependence graph for the critical section shown in Figure 5.11. Suppose that $Pot_W(A) = \{X, Y\}$, D is a deterministic predicate, and I is an indeterminate predicate. $May_W(I)$ and $Must_W(I)$ are both $\{X\}$, and $May_W(D)$ and $Must_W(D)$ are {X,Y} and {X} respectively. Because $May_W(I) \subseteq Must_W(I)$, A will be correctly identified as sequential

```
procedure Check-Reference(A)
    Compute-May_R(A)
    Compute-May_W(A)
    if state(root(A)) = deterministic and there is no coordination in A then
        Compute-Must_R(A)
        Compute-Must_W(A)
    endif
    for all n ∈ A such that state(n) ≠ deterministic do
        if May_W(n) ⊈ Must_W(n) or May_R(n) ⊈ Must_R(n) then Reference Nondeterministic
    endfor
end procedure
```

Algorithm 5.5: Detecting Reference Nondeterminism

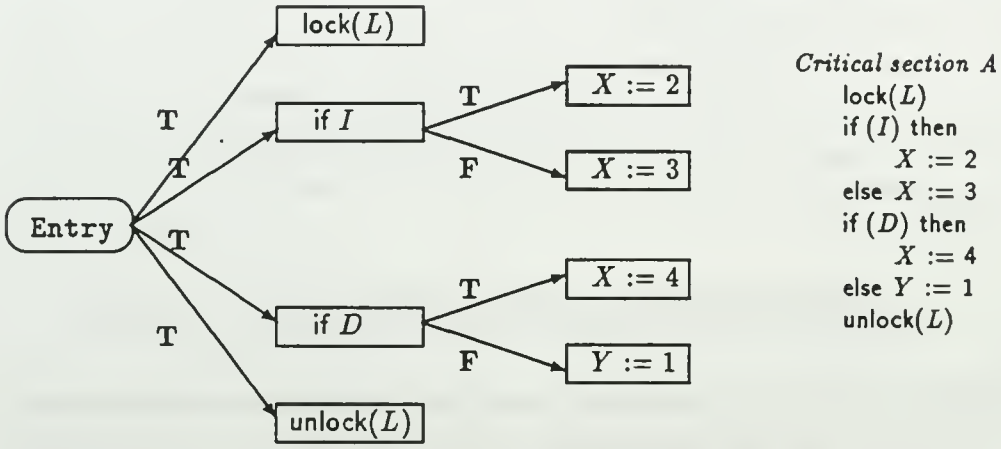


Figure 5.11: Reference Deterministic Program

deterministic. Since D is deterministic, the fact that $May_W(D) \not\subseteq Must_W(D)$ does not affect the detection of reference nondeterminism.

Theorem 19 *Every reference nondeterministic construct is detected by procedure* Check-Reference.

Proof. Suppose, for the sake of contradiction, some variable X with a reference nondeterministic use in critical section A is not detected by *Check-Reference*. We assume that the set of potentially access anomalies variables is correctly computed, and therefore, X must be in $Pot_R(A)$. If a use U of X occurs in some execution instances and not in others, U must be control dependent on a set of nodes $N = n_1 \ldots n_j$. There are several cases to consider:

1. All nodes in N are deterministic. Whether or not U occurs is independent of the critical section execution order.

2. Some node in N is not deterministic and is not in A. U cannot be in A since $\text{Must}_R(n) = \emptyset$ for all n in A.

3. All nodes in N that are not deterministic are in A. If A contains coordination then U cannot be in A since $\text{Must}_R(n) = \emptyset$ for all n in A. Otherwise, $X \in \text{Must}_R(n)$ for every n in N that is not determinate. Therefore, whether or not X is used is independent of the critical section execution order.

In all cases, a contradiction is reached. □

5.4 Heuristics for Nondeterministic Critical Sections

In order to guarantee that a program and input vector pair is anomaly free, an execution instance must be generated and analyzed for each possible POEG. Unfortunately, this task is generally intractable. For instance, 40,320 execution instances would have to be analyzed dynamically in order to guarantee that a very modest parallel program with 8 concurrent critical sections does not contain any access anomalies.

Because of the basic infeasibility of this goal, a dynamic anomaly detection system generally will not attempt to guarantee anomaly freedom. Instead, its goal is to ensure that false anomalies are not reported. In this case, alternative representations of critical sections can be used that make dynamic anomaly detection more effective. Access anomalies will not be "hidden" by coordination edges added to the POEG solely to reflect nondeterminism, the number of execution instances is decreased, and the structure of the POEG is simplified. Section 5.4.1 describes the approach of ignoring nondeterminism.

Alternatively, the effects of parallel, reference and sequential nondeterminism on a given statically detected potential anomaly can be analyzed. By doing so, properties can be guaranteed about certain classes of potential access anomalies. Section 5.4.2 describes the localization technique.

5.4.1 Ignoring Nondeterminism

When the requirement of detecting anomaly freedom is relaxed, certain types of nondeterminism can be ignored.

- Whether or not a given parallel construct is sequential nondeterministic affects the number of execution instances that must be modeled to guarantee anomaly freedom. However, as is shown by Theorem 18, sequential nondeterminism does not influence

the reliability of the unordered representation to detect anomalies *within* a given construct or cause false anomalies to be reported in subsequent code.

- It may be valid to assume that reference nondeterminism is always unintentional. There is no obvious use of reference nondeterminism that increases the power or clarity of a parallel program. Given this assumption, parallel constructs that are parallel deterministic but potentially reference nondeterministic, can be analyzed using an adaptive access anomaly representation. Initially, the unordered representation is used. However, if false anomalies are reported, the ordered representation is required to correctly model the reference nondeterministic behavior of the program.

Thus, it may be sufficient for a static analysis system to detect only parallel nondeterminism, which can be done relatively efficiently, and ignore sequential and reference nondeterminism.

In general, parallel nondeterminism cannot be simply ignored. As was illustrated by many previous examples, the ordered representation is often needed to correctly capture the semantics of useful critical section behavior. Fortunately, many important uses of critical sections exhibit "restricted" parallel nondeterminism. Parallel nondeterminism is restricted if the actions of a thread are not based on information about all of the threads that entered the critical section before it; rather they are based on information about a limited number of other threads. Knowledge about the semantics of parallel nondeterministic critical section coordination can be used to add only the minimal amount of ordering to the POEG, thereby exposing more access anomalies in any given execution instance and reducing the number of execution instances.

The remainder of this section describes two important examples of restricted parallel nondeterministic coordination: *Fetch&ϕ* and *working set coordination*. Table 5.1 summarizes the effects of parallel, reference, and sequential nondeterminism on access anomaly detection in programs with critical section coordination.

Fetch&ϕ Coordination

Fetch&ϕ is a class of read-modify-write hardware instructions provided on some architectures; most notably, the NYU Ultracomputer and IBM RP3 [Got88,PBG+85][4]. Fetch&ϕ indivisibly returns the current value of some variable X and then performs an associative, binary operation ϕ using X and a passed parameter. If hardware support is not provided, Fetch&ϕ can be implemented in software using critical sections as is shown in Figure 5.12. Whether provided by hardware or simulated in software, a Fetch&ϕ operation is correctly

[4]Other multiple operation instructions, such as Test&Set, can be analysed using the same techniques presented below for modeling Fetch&ϕ.

Conditions	*Properties*
Sequential Determinism Reference Determinism Parallel Determinism	The unordered representation will not report false anomalies. One execution instance is sufficient to prove anomaly freedom. There is no probe effect for detecting anomalies.
Sequential Nondeterminism Reference Determinism Parallel Determinism	The unordered representation will not report false anomalies. $N!$ execution instances required to prove the subsequent code anomaly freedom.
Sequential Determinism Reference Nondeterminism Parallel Determinism	An adaptive representation is used as follows: the unordered representation is initially used; if false access anomalies are reported then the ordered representation is used. $N!$ execution instances required to prove anomaly freedom for the ordered representation.
Sequential Determinism Reference Determinism Fetch&ϕ Coordination	The unordered representation will not report false anomalies if operations are uniform. $N!$ execution instances required to prove anomaly freedom.
Sequential Determinism Reference Determinism Work Set Coordination	The token based ordered representation can be used. $\frac{N}{2}!$ execution instances required to prove anomaly freedom.
Sequential Determinism Reference Determinism Parallel Nondeterminism	The ordered representation is used. $N!$ execution instances required to prove anomaly freedom.

Table 5.1: Summary of Heuristic Representations

modeled in dynamic access anomaly detection systems as a critical section.

Fetch&ϕ has proved to be very useful in writing parallel programs. To illustrate the power of Fetch&ϕ, consider the program in Figure 5.13 that uses the *self-service* paradigm [Got88]. In this paradigm, whenever a thread becomes idle, it assigns itself work by modifying a global variable. The global variable can be efficiently maintained by using Fetch&Add operations. In the program in Figure 5.13, each of the parallel iterates performs Fetch&Add operations to assign itself a series of indices of array A to process. Each iterate terminates when all 100 entries in A have been processed.

A set of Fetch&ϕ operations are *uniform* when all concurrent Fetch&ϕ operations to the same shared variables are identical (e.g. the same operation ϕ and parameter), such as in the program in Figure 5.13. There is no semantic ordering between threads that successively perform uniform Fetch&ϕ operations, since no information about the program state of threads that previously executed the critical section is known except perhaps how many other threads have already performed a Fetch&ϕ operation[5]. Thus,

[5]When exactly two concurrent threads perform Fetch&ϕ operations, knowing one other thread has

```
function Fetch&φ(X,i)
    lock(X.L)
    tmp := X.value
    X.value := φ(X.value, i)
    unlock(X.L)
    return tmp
end function
```

Figure 5.12: Fetch&ϕ

```
init(X, 1)
doall i := 1 to n
    r := Fetch&Add(X, 1)
    while r ≠ 100 do
        A[r] := f(r)
        r := Fetch&Add(X, 1)
    end while
end all
```

Figure 5.13: Self-service work assignment with Fetch&Add

no false anomalies will be reported when the unordered representation is used to model uniform Fetch&ϕ operations. The actions performed by a block can be dependent on the value returned by a Fetch&ϕ operation, however, and hence $N!$ execution instances are still required to guarantee that the program does not contain any access anomalies.

Work Set Coordination

In work set coordination, *tokens* are associated with shared variables that control access to those variables. A block b can access a shared variable only if it has the *token* associated with that variable. Once b obtains the token from the work set, it can access the data associated with that token without introducing any anomalies. When b finishes accessing a shared variable, it frees the associated token by adding it to the work set. This type of coordination may be exhibited in programs that use, for example, mailboxes, queues, or stacks or the Linda programming model [GCCC85][6].

To illustrate work set coordination, consider the program fragment in Figure 5.14. In this program a shared queue stores information about the entries in an array that have finished the initialization stage and are ready for the computation phase. After an

executed the critical section is equivalent to knowing the order of execution.

[6] We do not address the issue of detecting working set coordination, a problem posed by Emrath and Padua [EP88].

```
parallel case
    doall j := 1 to 3
        initialize A[j]
        lock(L)
        enqueue(j)
        unlock(L)
    endall
parallel
    doall j := 1 to 3
        lock(L)
        dequeue(k)
        unlock(L)
        process A[k]
    endall
end parallel case
```

Figure 5.14: Work Set coordination example

iterate initializes an entry $A[j]$, it enqueues the index j. When an iterate needs an entry to process, it dequeues the index of an initialized entry. The accesses performed in the initialization of an entry i never overlap with the processing of i because of the implicit coordination through the index i.

Correct access to the work set pool is obtained by enclosing all operations on it within critical sections. If the unordered critical section representation is used to model these critical sections, false anomalies can be reported. Anomalies are correctly detected when the ordered critical section representation is used; however, too much ordering is imposed, since the edges needed to represent implicit ordering from work set coordination are a subset of those in the ordered representation. A correct and efficient representation of work set coordination only orders blocks that exchange a token.

A work set implementation of critical sections requires that the dynamic anomaly detection system knows which critical sections add tokens to the pool and which extract tokens and lock covers must be used to model accesses to the pool. When a block B adds a token T to the pool, the block identifier of B is associated with T. When a block D deletes token T from the pool, a coordination edge is added to the POEG from B (the block whose identifier is associated with T) to D (the deleting block). Thus, only those blocks that have exchanged some state information that could be used in subsequent computations are ordered in the POEG. This ordering is generally of much finer granularity than the ordered critical section representation.

The benefits of properly modeling work set coordination are similar to those for properly modeling any coordination primitive. First, each block is ordered with only a minimal

set of other blocks. Therefore, the probability of detecting an access anomaly in any given execution instance is much higher. Second, the number of execution instances is greatly reduced. In general, there are $(\frac{N}{2})!$ rather than $N!$ different execution instances to consider.

The advantage of explicitly representing work set coordination can be seen by considering an execution instance of the program in Figure 5.14. Figure 5.15 shows the POEG

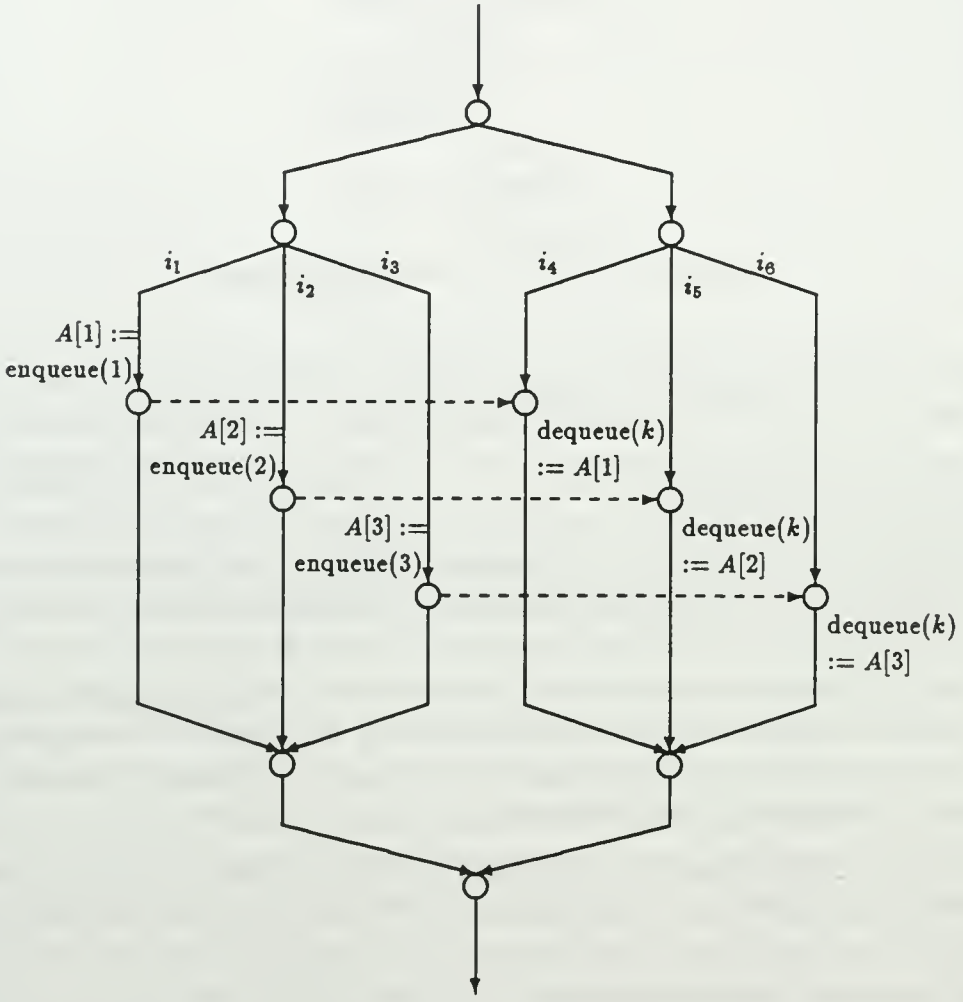


Figure 5.15: POEG for Work Set Coordination

for the execution instance in which iterates i_1, i_2 and i_3 respectively add tokens 1, 2 and 3 to the pool, and iterate i_4 deletes token 1, iterate i_5 deletes token 2, and iterate i_6 deletes token 3. The edge from i_2 to i_5 indicates that a token that i_2 added to the work set was removed by i_5.

First, the work set representation adds three coordination edges, resulting in each thread being ordered with one additional block. The ordered representation would add five coordination edges, so that each thread is ordered with five additional blocks. Second, by using the work set representation, only 6 execution instances must be analyzed to guarantee that the program in Figure 5.15 is anomaly free. In contrast, 720 execution instances would have to be considered if the ordered representation were used.

5.4.2 Localizing Nondeterminism

In general, static access anomaly detection is used to identify a set of potential access anomalies, and dynamic detection is then used to determine if these are actual access anomalies. Even if a parallel construct is nondeterministic, a given potential access anomaly in that construct can be independent of the execution order of its critical section. We say that an access anomaly is *invariant* if and only if neither of the conflicting accesses is parallel, sequential or reference nondeterministic[7]. This property is determined by producing a forward slice from every nondeterministic use and definition found using the algorithm described in Section 5.3

For example, consider the parallel nondeterministic program in Figure 5.16. Although

```
X := 0
doall i := 1 to 2
    Y := A[i]
    lock(L)
    X := X + 1
    j := X
    unlock(L)
    A[j] := Y × i
endall
```

Figure 5.16: Local affect of nondeterminism

this program is parallel nondeterministic, the potential access anomalies to Y are invariant and will occur regardless of the critical section execution order. The potential access anomalies to array A are not invariant, since the write of A is control dependent on j which is parallel nondeterministic. The access anomaly to $A[2]$ will only be reported in those execution instances in which iterate 2 enters the critical section before iterate 1.

Access anomalies that are invariant can be reliably detected even in the presence of parallel, reference and sequential nondeterminism. In particular, in the absence of other

[7]This is similar to the notion of *static anomalies* proposed by Allen and Padua for the hides relationship [AP87]. Because this work focuses on the nondeterminism stemming from critical section coordination, a more accurate classification is possible.

access anomalies, one execution instance is sufficient to guarantee that a POEG deterministic program with critical sections and input vector pair is invariant anomaly free. (This follows from the fact that in the absence of other access anomalies, every sequential, parallel and reference deterministic use or definition occurs in every execution instance.) Therefore, if every potential access anomaly in P is invariant, then one execution instance is sufficient to guarantee that P and I is anomaly free.

Chapter 6

Conclusions

This thesis addresses the issue of detecting a specific type of nondeterminism in shared memory parallel programs known as an *access anomaly*. An access anomaly occurs when an update to a shared variable X is concurrent with either a read of X or another update of X. Access anomalies are often bugs. Even when intentional, an access anomaly introduces nondeterminism which makes understanding and debugging parallel program very difficult. Thus, it is essential that access anomalies are isolated–not necessarily so that they can be eliminated, rather so that they are made explicit and their effect on program behavior can be considered. Unless access anomalies are detected and reported, we can have little confidence in the reliability of parallel programs.

The primary contributions of this thesis are as follows:

1. **Task Recycling Technique**
   A new on-the-fly access anomaly detection algorithm referred to as task recycling is presented in Chapter 3. Task recycling is a significant improvement over existing techniques in several ways. First, it supports a wide class of parallel programs. All previous techniques support only a subset of common parallel programming models. For instance, none supported the Ada tasking model.

   Second, task recycling is more efficient for many programs. It minimizes the cost per variable access (since this generally is assumed to be the most common operation), and requires an amount of space that is a function of the maximum parallelism in the program, rather than the total length of the execution.

   In addition, we show how the task recycling technique can be integrated with trace-and-replay and event-based debugging systems to improve their reliability.

2. **Empirical Measurements**
   While several on-the-fly access anomaly detection algorithms have been proposed, none had been implemented and little was known about the actual costs of detecting

anomalies in "real" parallel programs. To this end, Chapter 4 presents empirical measurements gathered from monitoring several scientific programs using the task recycling technique and an alternative technique, known as English-Hebrew labeling.

We show that program behavior that is common in parallel scientific codes allows for inexpensive detection of access anomalies. The actual overheads that are incurred (less that 350% slowdown for the benchmark programs) indicate that on-the-fly anomaly detection is efficient enough to be a viable debugging tool. Moreover, task recycling compares favorably with respect to English-Hebrew labeling, both in terms of time and space.

3. **Critical Section Nondeterminism**
   Chapter 5 presents a new technique for representing critical section coordination that greatly increases the effectiveness of dynamic anomaly detection. This representation can be used when the execution order of concurrent critical sections does not affect the execution of the rest of the program. (For instance, consider a program in which the critical sections are used to compute a maximum.) We present algorithms–based on static compiler optimization techniques–for detecting several different types of nondeterminism that can arise from critical section coordination. The absence of these types of nondeterminism determines when the the new representation of critical sections is reliable.

   The techniques used to analyze critical section coordination give insight into similar algorithms for other coordination primitives: for example, the Ada rendezvous. A better understanding of the behavior of critical section can also lead to more efficient trace-and-replay debugging and testing of parallel programs.

Bibliography

[ABC+88] Frances Allen, Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, and Vivek Sarkar. A Framework for Determining Useful Parallelism. *Proceedings of the ACM 1988 International Conference on Supercomputing*, pages 207–215, July 1988.

[AM85] Bill Appelbe and Charles E. McDowell. Anomaly Reporting–A Tool for Debugging and Developing Parallel Numerical Applications. In *Proceedings of the 1st International Conference on Supercomputers*, December 1985.

[AM88] Bill Appelbe and Charles E. McDowell. Developing Multitasking Applications Programs. In *Proceedings of the 21st Annual Hawaii International Conference on Systems Science*, pages 94–102, 1988.

[Amd67] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Conference*, volume 30, pages 483–485, 1967.

[AP87] Todd R. Allen and David A. Padua. Debugging Fortran on a Shared Memory Machine. In *Proceedings of the International Conference on Parallel Processing*, pages 721–717, August 1987.

[AS83] G.R. Andrews and F.B. Schneider. Concepts and Notions for Concurrent Programming. *Computing Surveys*, 15(1), March 1983.

[AU77] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[Axe86] T.S. Axelrod. Effects of Synchronization Barriers on Multiprocessor Performance. *Parallel Computing*, 3.:129–140, 1986.

[Bat88] Peter C. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 11–22, May 1988.

[BCF+88] Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, Vivek Sarkar, and David Shields. Automatic discovery of parallelism: A tool and an experiment. *ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 77–84., July 1988.

[Ber66] A. J. Bernstein. Program Analysis for Parallel Processing. *Transactions on Electronic Computers*, EC-15(5):757–762, October 1966.

[Ber88] Wayne Berke. ParFOR - A Structured Environment for Parallel FORTRAN. Technical Report Ultracomputer Report #137, New York University, April 1988.

[BH83] Bernd Bruegge and Peter Hibbard. Generalized Path Expressions: A High-Level Debugging Mechanism. *The Journal of Systems and Software*, 3:265–276, 1983.

[BHRB89] William Baxter and III Henry R. Bauer. The program dependence graph in vectorization. *Sixteenth ACM Principles of Programming Languages Symposium*, January 11-13 1989. Austin, Texas.

[BK89] Vasanth Balasundaram and Ken Kennedy. Compile-time Detection of Race Conditions in a Parallel Program. In *International Conference on Supercomputing*, June 1989.

[BKK+88] Vasanth Balasundaram, Ken Kennedy, Uli Kremer, Kathryn McKinley, and Jaspal Subhlok. PTOOL: A Systen for Static Analysis of Parallelism in Programs. Technical Report Rice COMP TR88-71, Rice University, June 1988.

[BKK+89] Vasanth Balasundaram, Ken Kennedy, Uli Kremer, Kathryn McKinley, and Jaspal Subhlok. The ParaScope Editor: An Interactive Parallel Programming Tool. Technical Report Rice COMP TR89-92, Rice University, May 1989.

[Bru85] Bernd Bruegge. *Adaptability and Portability of Symbolic Debuggers.* PhD thesis, Carnegie Mellon University, September 1985.

[BW83] Peter C. Bates and Jack C. Wileden. High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *The Journal of Systems and Software*, 3:255–264, 1983.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method for Computing Static Single Assignment Form. In *16th Annual ACM Symposium on the Principles of Programming Languages*, January 1989.

[CH73] R.H. Campbell and A.N. Haberman. The Specification of Process Synchronization by Path Expressions. Technical report, Carnegie-Mellon University, December 1973.

[CHH89] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic Generation of DAG Parallelism. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989.

[Cho89] Jongdeok Choi. *Parallel Program Debugging with Flowback Analysis.* PhD thesis, University of Wisconsin–Madison, 1989.

[CK87] David Callahan and Ken Kennedy. Analysis of Interprocedural Side Effects in a Parallel Programming Environment. In *Proceedings of the First International Conference on Supercomputing*, pages 139–171, June 1987.

[CKS90] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of Event Synchronization in a Parallel Programming Tool. In *Proceedings of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990.

[CMN88] Jong-Deok Choi, Barton P. Miller, and Robert Netzer. Techniques for Debugging Parallel Programs with Flowback Analysis, August 1988.

[CR84] Lori Clarke and Debra Richardson. Symbolic Evaluation–An Aid to Testing and Verification. In H.L. Hausen, editor, *Software Validation.* Elsevier Science Publication B.V. (North-Holland), 1984.

[CS88] David Callahan and Jaspal Subhlok. Static Analysis of Low Level Synchronization. In *Proceedings of the SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 100–111, May 1988.

[Cyt84] Ron Cytron. *Compile-time Scheduling and Optimization for Asynchronous Machines.* PhD thesis, University of Illinois at Urbana-Champaign, 1984.

[Dij65] E.W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *CACM*, 8(9):569, September 1965.

[Dij68] E. W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages.* Academic Press, 1968.

[Dij71] E.W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1:115–132, 1971.

[Dij75] E.W. Dijkstra. Guarded Commands, Non–Determinism and a Calculus for the Derivation of Programs. *CACM*, 18(8):453–457, August 1975.

[Dil50] R.P. Dilworth. A Decomposition Theorem for Partially Ordered Sets. *Annals of Mathematics*, 51(1):161–166, January 1950.

[Din89] Anne Dinning. A Survey of Synchronization Methods for Parallel Computers. *IEEE Computer*, pages 66–75, June 1989.

[DKH88] Laura Dillon, Richard Kemmerer, and Linda Harrison. An Experience with Two Symbolic Execution-Based Approaches to Formal Verification of Ada Tasking Programs. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 114–122, July 1988.

[DS90] Anne Dinning and Edith Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, March 1990.

[EGP89] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event Synchronization Analysis for Debugging Parallel Programs. In *Proceedings of Supercomputing '89*, pages 580–588, November 1989.

[EP88] Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminancy in Parallel Programs. In *Proceedings of the SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[Fid88] C. J. Fidge. Partial Orders for Parallel Debugging. In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–184, May 1988.

[FLMC88] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, 1988.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, July 1987.

[GCCC85] D. Gelenter, N. Carriero, S. Chandron, and S. Chang. Parallel Programming in Linda. In *Proceedings of the 1985 Parallel Processing Conference*, pages 255–264, August 1985.

[GG77] J.B. Goodenough and S.L. Gerhart. Towards a Theory of Testing: Data Selection Criteria. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 2, pages 44–79. Prentice-Hall, 1977.

[Got88] Allan Gottlieb. An Overview of the NYU Ultracomputer Project. In J.J. Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 25–95. Elsevier, 1988.

[GPJL88] M.D. Guzzi, D.A. Padua, J.P.Hoeflinger, and D.H. Lawrie. Cedar Fortran and Other Vector and Parallel Fortran Dialects. In *Proceedings of Supercomputing '88*, pages 114–121, May 1988.

[Hab75] A.N. Haberman. Path Expressions. Technical report, Carnegie-Mellon University, June 1975.

[HIKS] Michael Hind, Emmanual Ishbiah, Philippe Kanony, and Ed Schonberg. An Ada Implementation of Task Recycling. Unpublished Document.

[HK73] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ Algorithm of Maximum Matching in Bipartite Graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[HK88] Wenwey Hseush and Gail E. Kaiser. Data Path Debugging: Data Oriented Debugging for a Concurrent Programming Language. In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 236–246, May 1988.

[HK90] Wenwey Hseush and Gail Kaiser. Modeling Concurrency in Parallel Debugging . In *Proceedings of the Second ACM SIGPLAN Symposium on Parallel Programming*, pages 11–20, March 1990.

[HKMC89] Robert Hood, Ken Kennedy, and John M. Mellor-Crummey. On-the-fly Detection Of Access Anomalies, December 1989. Unpublished Document.

[HPR87] Susan Horwitz, Jan Prins, and Thomas Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. *Conf. Rec. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 146–157, January 1987.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *Proceedings of the Sigplan '88 Conference on Programming Language Design and Implementation*, 23(7):35–46, July 1988.

[KLC+86] D. J. Kuck, D. Lawrie, R. Cytron, A. Sameh, and Daniel Gajski. Cedar project. In D.H. Sharp N. Metropolis W.J. Worlton, editor, *Frontiers of Supercomputing*, pages 97–123. U. of CA Press, 1986.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[MC88] Barton P. Miller and Jong-Deok Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.

[McD88] Charles E. McDowell. Viewing Anomalous States in Parallel Programs. In *International Conference on Parallel Processing*, volume 2, pages 54–57, 1988.

[NM89] Robert Netzer and Barton P. Miller. Detecting Data Races in Parallel Program Executions, November 1989. Unpublished Document.

[NR88a] Itzhak Nudler and Larry Rudolph. Indeterminancy Considered Harmful, 1988. Unpublished document.

[NR88b] Itzhak Nudler and Larry Rudolph. Tools for the Efficient Development of Efficient Parallel Programs. In *First Israeli Conference on Computer System Engineering*, 1988.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. *Software Engineering Notes*, 9(3), 1984. Also appears in Proceedings of the ACM Symposium on Practical Programming Development Environments, Pittsburgh, PA, April, 1984, and in SIGPLAN Notices, Vol. 19, No 5, May, 1984.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype RP3: Introduction and Architecture. In *International Conference on Parallel Processing*, pages 764–771, 1985.

[Per63] Micha A. Perles. A Proof of Dilworth's Decomposition Theorem for Partially Ordered Sets. *Israel Journal of Mathematics*, 1(2):105–107, June 1963.

[PM86] Robert Perron and Craig Mundie. The Architecture of the Alliant FX/8 Computer. In *IEEE CompCon*, pages 390–393, 1986.

[Pra86] Vaughan Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33 –71, 1986.

[Sch89] Edith Schonberg. On-The-Fly Detection of Access Anomalies. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[Sel89] Rebecca Parsons Selke. A Rewriting Semantics for Program Dependence Graphs. In *16th Annual ACM Symposium on the Principles of Programming Languages*, January 11-13 1989. Austin, Texas.

[Seq89] Sequent Computer Systems. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Technical Publications, 1989.

[Sni88] Marc Snir. Private correspondence, 1988.

[SS86] Ed Schonberg and D. Shields. Prototype of Efficient Implementation: A Case Study Using SETL and C. In *Hawaii International Conference on System Sciences*, January 1986.

[Sta83] American National Standards Institute. Ada programming language military standard, January 1983. American National Standards Institute, ANSI/MIL-STD-1815A.

[Ste90] Guy L. Steele Jr. Making Asynchronous Parallelism Safe for the World. In *17th Annual ACM Symposium on the Principles of Programming Languages*, pages 218–231, January 1990.

[Sto88] Janice Stone. Debugging Concurrent Processes: A Case Study. In *Proceedings of the SIGPLAN Workshop on Parallel and Distributed Debugging*, May 1988.

[Tay83] Richard N. Taylor. A General-Purpose Algorithm for Analyzing Concurrent Programs. *Communications of the ACM*, 26(5):129–134, May 1983.

[TO80] Richard N. Taylor and Leon J. Osterweil. Anomaly Detection in Concurrent Software by Static Data Flow Analysis. *IEEE Transactions on Software Engineering*, SE-6(3), May 1980.

[Wei84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wei88] Stewart Weiss. A Formal Framework for the Study of Concurrent Program Testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 106–113, July 1988.

[WO80] Elaine J. Weyuker and Thomas J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions of Software Engineering*, pages 236–246, May 1980.

NYU COMPSCI TR-526 c.1
Dinning, Anne
Detecting nondeterminism i[n]
shared memory parallel
programs.

DATE DUE	BORROWER'S NAME

DATE DUE
